电子工程应用精讲系列

# Freescale系列单片机常用模块与综合系统设计实例精讲

张跃常 戴卫恒 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

## 内 容 简 介

全书以 Freescale 的 HCS08 主流芯片为写作对象，通过实例精讲的形式，详细介绍了 Freescale 单片机常用模块与综合系统设计的方法与技巧。全书共分 3 篇 24 章：第一篇为基础知识篇，简要介绍了 Freescale 单片机的硬件结构与工作原理、指令系统、C 语言开发基础及集成开发环境，引导读者入门；第二篇为常用模块设计实例篇，通过 18 个模块实例，详细介绍了 Freescale 单片机的各种开发技术和使用技巧，这些模块实例基础、实用，易学易懂，全部调试通过，几乎涵盖了 Freescale 单片机所有的开发技术；第三篇为综合系统设计实例篇，通过 2 个复杂项目，对前面的 Freescale 单片机常用模块进行了综合应用设计，经过此篇学习，读者的 Freescale 单片机设计能力将迅速提升，步入高级工程师的行列。

本书配有一张光盘，包含了全书所有实例的硬件原理图和程序源代码，方便读者学习和使用。本书的读者对象为 Freescale 单片机的初、中级用户，特别适合计算机、自动化、电子及硬件等相关专业的大学生及从事 Freescale 单片机开发的工程师阅读使用。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

**图书在版编目（CIP）数据**

Freescale 系列单片机常用模块与综合系统设计实例精讲 / 张跃常，戴卫恒编著. —北京 ：电子工业出版社, 2010.7（电子工程应用精讲系列）
ISBN 978-7-121-11090-0

Ⅰ. ①F… Ⅱ. ①张… ②戴… Ⅲ. ①单片微型计算机－系统设计 Ⅳ. ①TP368.1

中国版本图书馆 CIP 数据核字(2010)第 109806 号

责任编辑：张月萍
印　　刷：北京天宇星印刷厂
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：23　字数：588.8 千字
印　　次：2010 年 7 月第 1 次印刷
印　　数：4000 册　定价：49.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

# 前　言

Freescale 系列单片机是飞思卡尔公司开发的旗舰产品。该公司原为摩托罗拉的半导体部，是全球最大的半导体公司之一，在我国的北京、上海、苏州和天津设有 4 个研发中心，在国内二十余所重点大学校园中建立了 Freescale 单片机实验室与培训合作机构。Freescale 系列单片机逐渐成为全球电子产品市场的生力军。但目前市场上 Freescale 单片机类的图书比较少，而且以介绍编程语言和基础原理为主，而对应用开发的实例内容涉及甚少，从常用模块到综合应用系统的提高实例更是处于空白状态。本书的出版正是为了填补这种空白。

## 本书内容安排

全书基于 Freescale 的主流芯片 HCS08 而编写，通过实例精讲的形式，详细介绍了 Freescale 单片机常用模块与综合系统设计的方法与技巧。全书共分 3 篇 24 章，具体如下：

第一篇为基础知识篇，简要介绍了 Freescale 单片机的硬件结构与工作原理、指令系统、C 语言开发基础及常用开发工具，引导读者熟悉 Freescale 单片机的硬件特点，掌握编程入门。

第二篇为常用模块设计实例篇，结合 18 个模块实例，详细介绍了 Freescale 单片机的各种开发技术和使用技巧。这些模块实例基础、实用，易学易懂，全部调试通过，读者学习后举一反三，可以掌握 Freescale 单片机绝大多数的开发技术。

第三篇为综合系统设计实例篇，通过 2 个复杂的项目，对前面的 Freescale 单片机常用模块进行综合应用设计。经过此篇的学习，读者的 Freescale 单片机设计能力可以迅速提升，继而步入高级工程师的行列。

本书配有一张光盘，包含了全书所有实例的硬件原理图和程序源代码。

## 本书读者对象

本书主要针对 Freescale 单片机的初、中级用户编写，特别适合计算机、自动化、电子及硬件等相关专业的大学生及从事 Freescale 单片机开发的工程师阅读使用。

## 本书优势和特色

与同类书相比，本书主要具备如下的一些优势和特色：

（1）本书内容系统丰富、从简到难，将基础知识与大量实例相结合，边讲边练，使读者学习轻松、上手容易。

（2）本书的 18 个 Freescale 单片机常用模块实例基础、典型、实用，几乎涵盖了所有的 Freescale

单片机开发技术，为读者提供了便捷的学习途径。

（3）最后安排了 2 个综合系统实例，对 Freescale 单片机的基本知识和常用模块进行了综合应用，有利于读者举一反三，实现从入门到精通的学习目的。

（4）本书在讲解方式上，不但详细介绍了 Freescale 单片机的硬件接口设计和模块化编程，而且提供了综合系统设计思路，并对实例的所有程序代码做了详细注释，帮助读者理解和巩固知识点，深化各项开发技术。

本书主要由张跃常、戴卫恒编写，另外参与编写的人员还有：赵汶、唐清善、邱宝良、李宁宇、严剑忠、黄小宽、付军鹏、张广安、贾素龙、金平、徐春林、谢正义、郑贞平，在此一并向他们表示感谢！

由于时间仓促，再加之作者的水平有限，书中难免存在一些不足之处，欢迎广大读者批评指正。

编著者

2010.4

# 目　录

## 第一篇　基础知识篇


第 1 章　Freescale 单片机的硬件内部结构 ..... 2
1.1　HCS08 系列单片机的简介 ..... 2
1.1.1　HCS08 系列 MCU 的基本特征 .. 2
1.2　MC9S08JM60 单片机的硬件结构 ... 3
1.2.1　MC9S08JM60 的 CPU ..... 3
1.2.2　JM60 存储器的组织结构 ..... 5
1.2.3　JM60 的复位、中断和系统配置 .. 6
1.2.4　多功能时钟发生器（MCG）模块 ..... 12
1.2.5　I/O 端口 ..... 17
1.2.6　KBI 模块 ..... 29
1.2.7　定时器/PWM 模块 ..... 31
1.2.8　IIC 模块 ..... 35
1.2.9　FLASH 模块 ..... 39
1.2.10　SCI 模块 ..... 42
1.2.11　SPI 模块 ..... 49
1.2.12　ADC 模块 ..... 53
1.2.13　USB 模块 ..... 58
1.3　本章总结 ..... 68
第 2 章　HCS08 单片机的指令系统 ..... 69
2.1　HCS08 内核的介绍 ..... 69
2.1.1　HCS08 内核特点 ..... 69
2.1.2　HCS08 CPU 寄存器 ..... 69
2.2　寻址方式 ..... 72
2.3　HCS08 指令系统 ..... 74
2.3.1　数据传送指令 ..... 74
2.3.2　算术运算指令 ..... 77
2.3.3　逻辑运算指令 ..... 81
2.3.4　位操作指令 ..... 81
2.3.5　移位类指令 ..... 82
2.3.6　程序转移和控制类指令 ..... 83
2.4　本章总结 ..... 88
第 3 章　HCS08 单片机的 C 语言程序设计 ..... 89
3.1　单片机 C 语言设计基础 ..... 89
3.1.1　数据结构 ..... 89
3.1.2　运算符与表达式 ..... 108
3.1.3　程序结构与函数 ..... 112
3.1.4　流程控制语句 ..... 120
3.2　HCS08 单片机 C 语言的扩展语法 ..... 128
3.2.1　基本数据类型 ..... 128
3.2.2　变量定义 ..... 129
3.2.3　变量定义修饰符 ..... 130
3.2.4　全局变量和局部变量 ..... 131
3.2.5　位定义和访问 ..... 132
3.2.6　中断服务程序定义 ..... 133
3.2.7　内嵌汇编和混合调用 ..... 134
3.3　本章总结 ..... 137

第 4 章　HCS08 单片机的开发工具 ... 138
4.1　CodeWarrior for Microcontrollers V6.2 的安装 ... 138
4.1.1　系统需求 ... 138
4.1.2　CodeWarrior 的安装与卸载 ... 139
4.2　CodeWarrior for Microcontrollers V6.2 功能使用 ... 140
4.2.1　集成环境与功能设置 ... 140
4.3　CodeWarrior 开发 HCS08 单片机的过程 ... 143
4.3.1　建立和打开工程 ... 143
4.3.2　编译和链接 ... 147
4.3.3　软件仿真调试方法 ... 147
4.3.4　BDM 调试方法 ... 149
4.4　本章总结 ... 150

# 第二篇　常用模块设计实例篇

第 5 章　通用 IO 模块：流水灯设计实例 ... 152
5.1　实例内容说明 ... 152
5.2　设计思路分析 ... 153
5.3　硬件电路设计 ... 153
5.3.1　电源电路 ... 153
5.3.2　复位电路 ... 153
5.3.3　BDM 仿真调试电路 ... 154
5.3.4　流水灯电路 ... 154
5.3.5　单片机电路 ... 154
5.4　软件设计与代码 ... 155
5.4.1　软件设计流程 ... 155
5.4.2　初始化程序 ... 155
5.4.3　主程序 ... 157
5.5　实例总结 ... 157
第 6 章　通用 IO 模块：语音播放系统设计实例 ... 158
6.1　实例内容说明 ... 158
6.2　设计思路分析 ... 159
6.3　硬件电路设计 ... 159
6.3.1　ISD2560 介绍 ... 159
6.3.2　语音电路 ... 160
6.3.3　单片机电路 ... 161
6.4　软件设计与代码 ... 161
6.4.1　软件设计流程 ... 161
6.4.2　初始化程序 ... 162
6.4.3　ISD2560 录音、放音子程序 ... 164
6.4.4　主程序 ... 164
6.5　实例总结 ... 166
第 7 章　液晶显示模块：温度测量系统设计实例 ... 167
7.1　实例内容说明 ... 167
7.2　设计思路分析 ... 168
7.3　硬件电路设计 ... 168
7.3.1　DS18B20 介绍 ... 169
7.3.2　温度传感器电路 ... 169
7.3.3　液晶显示电路 ... 170
7.3.4　单片机电路 ... 170
7.4　软件设计与代码 ... 171
7.4.1　软件设计流程 ... 171
7.4.2　温度测量程序 ... 171
7.4.3　LCD 显示程序 ... 173
7.4.4　主程序 ... 174
7.5　实例总结 ... 175
第 8 章　液晶显示模块：实时时钟设计实例 ... 176
8.1　实例内容说明 ... 176
8.2　设计思路分析 ... 177
8.3　硬件电路设计 ... 177

8.3.1 按键和 LED 电路 177
8.3.2 液晶显示电路 178
8.3.3 单片机电路 178
8.4 软件设计与代码 178
8.4.1 软件设计流程 178
8.4.2 时钟子程序 179
8.4.3 LCD 显示程序 181
8.4.4 主程序 182
8.5 实例总结 185
第 9 章 KBI 模块：扫描键盘设计实例 186
9.1 实例内容说明 186
9.2 设计思路分析 187
9.3 硬件电路设计 188
9.3.1 按键电路 188
9.3.2 LED 显示电路 188
9.3.3 单片机电路 189
9.4 软件设计与代码 190
9.4.1 软件设计流程 190
9.4.2 键盘扫描子程序 190
9.4.3 LED 显示程序 191
9.4.4 主程序 192
9.5 实例总结 193
第 10 章 定时器模块：用定时器控制蜂鸣器发声 194
10.1 实例内容说明 194
10.2 设计思路分析 195
10.2.1 蜂鸣器发声原理 195
10.2.2 蜂鸣器发声控制 195
10.3 硬件电路设计 196
10.3.1 蜂鸣器电路 196
10.3.2 单片机电路 196
10.4 软件设计与代码 196
10.4.1 软件设计流程 196
10.4.2 外设初始化程序 197
10.4.3 定时中断服务程序 197
10.4.4 蜂鸣器控制子程序 198
10.4.5 主程序 199
10.5 实例总结 199
第 11 章 定时器模块：电子钟系统设计实例 200
11.1 实例内容说明 200
11.2 设计思路分析 201
11.3 硬件电路设计 202
11.3.1 闹钟报警提示电路 202
11.3.2 按键电路 202
11.3.3 LCD 显示电路 203
11.3.4 单片机电路 203
11.4 软件设计与代码 204
11.4.1 软件设计流程 204
11.4.2 初始化程序 204
11.4.3 按键扫描程序 205
11.4.4 LCD 显示程序 206
11.4.5 主程序 208
11.5 实例总结 209
第 12 章 存储器模块：FLASH 编程实现 210
12.1 实例内容说明 210
12.2 设计思路分析 211
12.2.1 寄存器介绍 211
12.2.2 FLASH 编程和擦除命令的执行顺序 214
12.3 硬件电路设计 215
12.4 软件设计与代码 215
12.4.1 软件设计流程 215
12.4.2 FLASH 操作子程序 216
12.4.3 主程序 217
12.5 实例总结 218

第 13 章　AD 模块：加速度传感器系统设计实例 ..... 219
13.1　实例内容说明 ..... 219
13.2　设计思路分析 ..... 220
13.3　硬件电路设计 ..... 221
13.3.1　MMA1220D 和 MMA1260D 的介绍 ..... 221
13.3.2　加速度传感器电路 ..... 222
13.3.3　LCD 显示电路 ..... 222
13.3.4　单片机电路 ..... 223
13.4　软件设计与代码 ..... 224
13.4.1　软件设计流程 ..... 224
13.4.2　A/D 采样子程序 ..... 224
13.4.3　加速度传感器子程序 ..... 225
13.4.4　LCD 显示程序 ..... 228
13.4.5　主程序 ..... 228
13.5　实例总结 ..... 230
第 14 章　AD 模块：烟雾检测系统设计实例 ..... 231
14.1　实例内容说明 ..... 231
14.2　设计思路分析 ..... 232
14.3　硬件电路设计 ..... 232
14.3.1　光电传感器电路 ..... 232
14.3.2　报警电路 ..... 233
14.3.3　单片机电路 ..... 233
14.4　软件设计与代码 ..... 234
14.4.1　软件设计流程 ..... 234
14.4.2　AD 采样子程序 ..... 235
14.4.3　定时程序 ..... 235
14.4.4　主程序 ..... 236
14.5　实例总结 ..... 237
第 15 章　PWM 模块：寻迹智能小车控制系统设计实例 ..... 238
15.1　实例内容说明 ..... 238
15.2　设计思路分析 ..... 239
15.2.1　PWM 基本原理 ..... 239
15.2.2　实现方法 ..... 240
15.3　硬件电路设计 ..... 240
15.3.1　H 形桥式电路 ..... 240
15.3.2　光电传感器电路 ..... 242
15.3.3　电源电路 ..... 242
15.3.4　单片机电路 ..... 242
15.4　软件设计与代码 ..... 243
15.4.1　软件设计流程 ..... 243
15.4.2　传感器采样处理子程序 ..... 244
15.4.3　智能车移动子程序 ..... 245
15.4.4　主程序 ..... 248
15.5　实例总结 ..... 248
第 16 章　通用串口通信模块：单片机与 PC 通信设计实例 ..... 249
16.1　实例内容说明 ..... 249
16.2　设计思路分析 ..... 250
16.2.1　电平转换芯片 ..... 250
16.3　硬件电路设计 ..... 250
16.3.1　MAX3232 芯片介绍 ..... 250
16.3.2　串口电路 ..... 251
16.3.3　单片机电路 ..... 251
16.4　软件设计与代码 ..... 252
16.4.1　软件设计流程 ..... 252
16.4.2　串口初始化程序 ..... 253
16.4.3　串口中断服务程序 ..... 253
16.4.4　主程序 ..... 255
16.5　实例总结 ..... 255
第 17 章　RS-485 总线通信模块：现场监测系统设计实例 ..... 256
17.1　实例内容说明 ..... 256
17.2　设计思路分析 ..... 257
17.2.1　RS-485 总线 ..... 257
17.2.2　RS-485 驱动芯片选取 ..... 258
17.3　硬件电路设计 ..... 258
17.3.1　SN75176 芯片介绍 ..... 258

17.3.2 232/485 转接卡 259
17.3.3 带 485 接口的单片机系统 259
17.4 软件设计与代码 260
17.4.1 软件设计流程 260
17.4.2 串口通信子程序 261
17.4.3 485 通信收发子程序 262
17.4.4 主程序 264
17.5 实例总结 265
第 18 章 I²C 总线通信模块：串行存储器接口设计实例 266
18.1 实例内容说明 266
18.2 设计思路分析 266
18.3 硬件电路设计 268
18.3.1 AT24C02 芯片的介绍 268
18.3.2 串行存储器电路 269
18.3.3 单片机电路 269
18.4 软件设计与代码 270
18.4.1 软件设计流程 270
18.4.2 I²C 模块初始化程序 271
18.4.3 AT24C02 的读/写子程序 271
18.4.4 主程序 274
18.5 实例总结 275
第 19 章 I²C 总线通信模块：I²C 主从通信系统设计实例 276
19.1 实例内容说明 276
19.2 设计思路分析 277
19.3 硬件电路设计 278
19.4 软件设计与代码 279
19.4.1 软件设计流程 279
19.4.2 I²C 程序 279
19.4.3 主程序 284
19.5 实例总结 287
第 20 章 SPI 总线通信模块：电量监测仪设计实例 288
20.1 实例内容说明 288
20.2 设计思路分析 289
20.2.1 SPI 协议 289
20.2.2 电能计量芯片的选择 291
20.3 硬件电路设计 291
20.3.1 CS5460A 芯片的介绍 291
20.3.2 电量信号采集电路 293
20.3.3 显示单元电路 294
20.3.4 单片机系统 294
20.4 软件设计与代码 295
20.4.1 软件设计流程 295
20.4.2 CS5460A 的操作 295
20.4.3 SPI 的实现 296
20.4.4 CS5460A 操作程序的实现 298
20.4.5 主程序 301
20.5 实例总结 302
第 21 章 SPI 总线通信模块：SD 卡数据存储系统设计实例 303
21.1 实例内容说明 303
21.2 设计思路分析 304
21.3 硬件电路设计 306
21.3.1 SD 卡接口电路 306
21.3.2 单片机系统电路 307
21.4 软件设计与代码 308
21.4.1 软件设计流程 308
21.4.2 SD 卡读/写子程序 308
21.4.3 主程序 312
21.5 实例总结 313
第 22 章 USB 串行总线模块：数据传输系统设计实例 314
22.1 实例内容说明 314
22.2 设计思路分析 315
22.3 硬件电路设计 317
22.3.1 USB 电路 317
22.3.2 加速度传感器电路 318
22.3.3 人机接口电路 318

22.3.4 单片机系统 318
22.4 软件设计与代码 319
22.4.1 软件设计流程 319
22.4.2 USB 模块初始化程序 319
22.4.3 USB 模块的相关程序 320
22.4.4 主程序 326
22.5 实例总结 326

# 第三篇 综合系统设计实例篇

第 23 章 出租车计价器系统设计实例 328
23.1 实例内容说明 328
23.2 设计思路分析 329
23.3 硬件电路设计 330
23.3.1 里程计算单元 330
23.3.2 显示电路 330
23.3.3 按键电路 331
23.3.4 单片机系统 331
23.4 软件设计与代码 332
23.4.1 软件设计流程 332
23.4.2 硬件层程序 332
23.4.3 应用程序接口层程序 334
23.4.4 应用层主程序 335
23.5 实例总结 344
第 24 章 手持式超声波测距仪系统设计实例 345
24.1 实例内容说明 345
24.2 设计思路分析 346
24.3 硬件电路设计 346
24.3.1 接收单元 346
24.3.2 发送单元 347
24.3.3 按键电路 347
24.3.4 显示单元 348
24.3.5 单片机系统 348
24.4 软件设计与代码 348
24.4.1 软件设计流程 348
24.4.2 超声波发送子程序 349
24.4.3 超声波接收子程序 350
24.4.4 按键扫描子程序 352
24.4.5 测距仪主程序 355
24.5 实例总结 356

# 第一篇

# 基础知识篇


- 第 1 章　Freescale 单片机的硬件内部结构
- 第 2 章　HCS08 单片机的指令系统
- 第 3 章　HCS08 单片机的 C 语言程序设计
- 第 4 章　HCS08 单片机的开发工具

# 第 1 章

# Freescale 单片机的硬件内部结构

作为本书的第 1 章，我们将主要介绍 Freescale 单片机的硬件内部结构，使读者对 Freescale 单片机有一个简单的认识。Freescale 单片机系列种类比较多，其中的 08 系列单片机由于集成度高、外围设备丰富、稳定性高、开发周期短、成本低、型号多种多样、兼容性好而被广泛应用。本章将在介绍 HCS08 系列单片机的基本特征的基础上，以 MC9S08JM60 为例讲述 HCS08 系列单片机的基本结构和特点。

## 1.1 HCS08 系列单片机的简介

Freescale 公司于 2004 年之后推出的 HCS08 系列 MCU 是高性能与低功耗的完美结合，其性能可与许多 16 位 MCU 相当。相对于 Freescale 公司单片机的其他系列，HCS08 更适合低功耗、高性能的使用环境，如便携式电子产品。HCS08 是 HC08 系列的一个有益补充。

### 1.1.1 HCS08 系列 MCU 的基本特征

#### 1. 多种低功耗模式

HCS08 系列具有 wait 模式和 3 种 stop 模式。

wait 模式使单片机进入等待状态，串行口、A/D、时钟等的中断均可唤醒。

stop 模式使系统时钟停止，电压调节器处于备用状态，用户可根据实际需要选择相应的 stop 模式。

stop1：内部电路处于完全掉电状态，以达到最大的省电模式。

stop2：内部电路部分处于掉电状态，内存的数据仍然保存。

stop3：所有的内部电路处于待机状态，可快速恢复到正常工作模式。

2. 更高的工作频率

工作电压为 2.1V 时，CPU 总线的最高频率为 20MHz；工作电压为 1.8V 时，CPU 总线的最高频率为 8MHz。

3. 具有可编程的片内时钟发生器

具有温度和电压补偿（典型漂移小于 2%）的片内可编程内部时钟发生器，能进一步减少系统成本。也可通过用户定义采用外接时钟。

4．新一代的内存技术

通过第三代 0.25 微米的内存技术提供应用程序的重复读写和数据存储能力。

5. 集成片上调试器

HCS08 系列新产品中还集成了片上调试器（BDM），实现了片上仿真、模拟等功能，极大地减少了开发工具的成本。在 16 位和 32 位 MCU 中集成 BDM 是很常见的，但在 8 位 MCU 中集成 BDM 还是第一次。

6. 资源丰富

RAM 内存高达 4KB，FLASH 存储器容量可达 60KB，高度集成了 4 个串行通信端口（SCI1、SCI2、SPI、IIC），最多达 8 个定时器、8 通道的 10 位 A/D 转换器。

# 1.2　MC9S08JM60 单片机的硬件结构

## 1.2.1　MC9S08JM60 的 CPU

MC9S08JM60 为高性价比、低功耗的 8 位微处理器 HCS08 家族中的一员，其功能及主要特性如下：

- 8 位 48MHz HCS08 CPU。
- 在整个工作电压和温度范围内可读取/擦除/编程的 60KB FLASH，有 4KB 的 RAM，对 RAM 及 FLASH 中的数据有保护机制。
- 支持 32 个中断/复位源。
- 等待模式及两种超低功耗的停止模式。
- 多功能的时钟生成器，提供包括 FLL、PLL、内部参考时钟、外部参考时钟等多种时钟源选项。
- 具有系统保护特性：
  - 计算机正常工作的看门狗（COP）复位；
  - 低电压检测复位或中断；
  - 非法操作码检测复位；
  - FLASH 块保护。

- 片上外部设备：
  USB 设备模块；
  模拟比较器；
  12 通道 12 位模数转换器（ADC）；
  两个全双工异步串行通信接口（SCI）；
  最高可达 100Kbps 的串行扩展总线（IIC）；
  两个全双工或单线同步串行外设接口（SPI）；
  高达 8 通道的定时器/PWM 模块。
- 51 个双向 I/O 口、一个单向输入口、一个单向输出口及 8 个键盘中断。
- 具有单线的背景调试接口、端点在线调试、在线仿真器（ICE）调试模块，8 级深度 FIFO 用于保存流程变化地址和单事件数据。

图 1-1 所示为 MC9S08JM60（以下简称为 JM60）的内部结构框图。

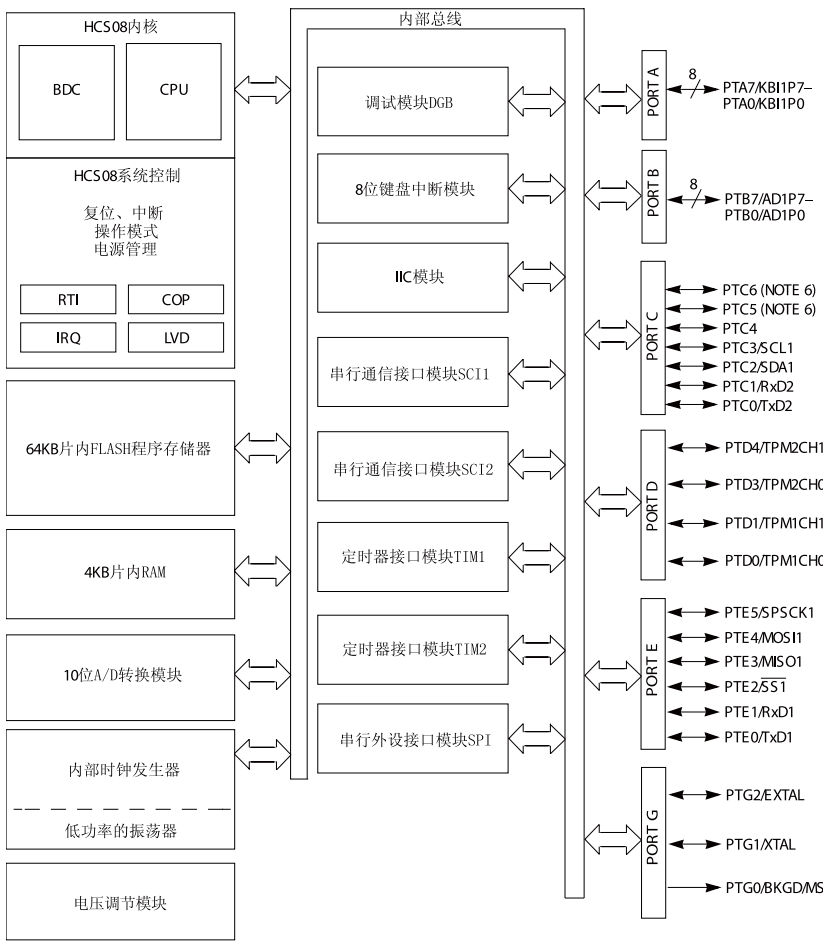



图 1-1　MC9S08JM60 结构框图

## 1.2.2　JM60 存储器的组织结构

JM60 存储器的映像图如图 1-2 所示。




图 1-2　JM60 的存储映像图

**1. RAM区域**

JM60 的内存 RAM 区容量达到 4KB，其地址范围是：＄0080~＄107F。相对于 GP32 的 512B，JM60 的 RAM 是相当丰富的资源，这对于完成一些对内存需求较大的嵌入式应用，是一个相当不错的选择。例如嵌入式网络系统、RTOS 等应用。

在这个区域中，有 128B 的 RAM 属于第 0 页，即＄0080~＄00FF，这个地址范围寻址方便，合理分配使用这些内存地址对于实际编程有很大帮助。

**2. FLASH区域**

JM60 有两个 FLASH 区域：从＄1080 到＄17FF 之间的 1920B 和从＄182C 到＄FFFF 之间的 59348B。

**3. 寄存器映像**

JM60 的寄存器有 3 个部分，包括：

- 直接页寄存器：$0000~$007F。
- 高端页寄存器：$1800~$182B。这个寄存器区放置了一些不常用的设置项。
- 非易失性寄存器：$FFB0~$FFBF。这个区位于 FLASH 存储区域中，对寄存器值的改变方法和其他位置的 FLASH 的擦写方法相同。这个区共 16B，前 8B（$FFB0~$FFB7）是访问 FLASH 的“后门钥匙”，详细解释将在后面章节的 FLASH 保密性部分阐述。$FFB8~$FFBC 是没有使用的保留部分，$FFBE 是厂家放置 ICG 修正值的位置。$FFBD-NVPROT、$FFBF-NVOPT 这两个值在 MCU 复位时分别赋给高端寄存器页的 FPRO（$1824）和 FOPT（$1821）。

### 1.2.3 JM60 的复位、中断和系统配置

**1. 复位**

复位使控制器进入初始状态。大部分状态和控制寄存器被迫使用初始值，并从复位向量（0xFFFE:0xFFFF）上载程序计数器。I/O 初始化配置为没有上拉电阻高阻抗输入，片上外围模块被禁止，中断被禁止。初始化程序可对堆栈指针（SP）和系统控制进行初始化。SP 在复位时强制设为 0x00FF。

JM60 系列有 7 个复位源：

- 上电复位（POR）
- 低电压检测（LVD）
- 计算机操作正常模块（COP）定时器
- 非法操作码检测（ILOP）
- 系统后台调试强制复位
- 外部复位引脚
- 时钟生成器锁定丢失和时钟复位丢失（LOC）

与系统复位相关的寄存器主要有系统复位状态寄存器（SRS）和系统后台调试强制复位寄存器（SBDFR）。SRS 寄存器为只读寄存器，通过读取标志位可以知道引起上一次复位的复位源。向 SRS 寄存器写入任意值可以使 COP 计数器清零。SBDFR 寄存器为只写寄存器，而且仅有最后一位（D0）可以写入。可以使用串行后台命令，如 WRITE_BYTE，使外部调试主机强制进行目标系统复位。该位不能从用户程序中写入。读该寄存器总是返回 0x00。

SRS 寄存器的直观描述如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | POR | PIN | COP | ILOP | 0 | LOC | LVD | |
| | 写 | 写入任意值可以使 COP 计数器清零 | | | | | | | |
| 复位 | | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

其中：

POR 是上电复位标志位。

由于在上电复位的过程中，电压值会出现低于 LVD 阈值的情况，因此，上电复位后 LVD 也会被置 1。

PIN 是外部引脚/RESET 的复位标志。

COP 是看门狗复位标志位。

ILOP 是非法操作码检测复位标志位。

LOC 是时钟丢失复位标志位。

LVD 是低电压检测复位标志。

**2. 中断**

程序在执行过程中，允许外部或内部事件通过硬件或软件中断当前程序，而转向中断服务程序中去，完成中断服务程序后，CPU 继续原来被中断的程序，这样的过程被称为中断过程。能产生中断的事件被称为中断源。软件中断是由程序指令触发的，而由诸如引脚电平的变化或定时器溢出等硬件事件触发的中断为硬件中断。在执行中断服务程序（ISR）前，当前的 CPU 状态和寄存器被保存，而在执行中断服务程序后，保存的 CPU 状态将被恢复。这样程序可以从中断前的位置重新开始执行。

当某一中断事件发生时，与该事件相对应的只读标志位将被置位。如果该中断未被使能，则 CPU 不会响应该中断，除非相应的中断使能位被置 1，并且 CCR 中的 I 位被清零，才能允许 CPU 响应中断。

当 CPU 接收到有效的中断请求时，它会在响应该中断前先完成当前指令，然后按顺序执行以下过程：

- 在堆栈上保存当前 CPU 寄存器。
- 置 CCR 中的 I 位，屏蔽中断。
- 获取当前被挂起的所有待处理中断事件中具有最高优先级的中断的中断向量。
- 预取中断向量地址的前 3 个字节排列到指令队列中。

当 CPU 响应中断时，会自动设置 I 位以避免嵌套的出现。在正常情况下，当中断服务程序执行结束后，CCR 从堆栈中弹出，I 位恢复为 0。

中断服务程序以中断恢复（RTI）指令作为结束。RTI 指令使先前保存的 CCR、A、X、PC 寄存器的值从堆栈中恢复。

表 1-1 为 JM60 的中断向量表，向量号越小，优先级越高。

表 1-1　JM60 的中断向量表

| 向 量 号 | 地　　址 | 向 量 名 | 中 断 源 |
|---|---|---|---|
| 31~30 | 0xFFC0~0xFFC3 | 未使用的向量空间 | |
| 29 | 0xFFC4~0xFFC5 | $V_{rtc}$ | 实时计数器中断 |
| 28 | 0xFFC6~0xFFC7 | $V_{iic}$ | iic 控制 |
| 27 | 0xFFC8~0xFFC9 | $V_{acmp}$ | 模拟比较器 |
| 26 | 0xFFCA~0xFFCB | $V_{adc}$ | ADC 转换 |

续表

| 向 量 号 | 地　址 | 向 量 名 | 中 断 源 |
|---|---|---|---|
| 25 | 0xFFCC~0xFFCD | $V_{keyboard}$ | 键盘中断 |
| 24 | 0xFFCE~0xFFCF | $V_{sci2tx}$ | SCI2 发送 |
| 23 | 0xFFD0~0xFFD1 | $V_{sci2rx}$ | SCI2 接收 |
| 22 | 0xFFD2~0xFFD3 | $V_{sci2err}$ | SCI2 错误 |
| 21 | 0xFFD4~0xFFD5 | $V_{sci1tx}$ | SCI1 发送 |
| 20 | 0xFFD6~0xFFD7 | $V_{sci1rx}$ | SCI1 接收 |
| 19 | 0xFFD8~0xFFD9 | $V_{sci1err}$ | SCI1 错误 |
| 18 | 0xFFDA~0xFFDB | $V_{tpm2ovf}$ | TPM2 溢出 |
| 17~16 | 0xFFDC~0xFFDF | $V_{tpm2ch1\sim Vtpm2ch0}$ | TPM2 通道 1~0 |
| 15 | 0xFFE0~0xFFE1 | $V_{tpm1ovf}$ | TPM1 溢出 |
| 14~9 | 0xFFE2~0xFFED | $V_{tpm1ch5\sim Vtpm1ch0}$ | TPM1 通道 5~0 |
| 8 | 0xFFEE~0xFFEF | 预留 | |
| 7 | 0xFFF0~0xFFF1 | $V_{usb}$ | USB 状态 |
| 6 | 0xFFF2~0xFFF3 | $V_{spi2}$ | SPI2 |
| 5 | 0xFFF4~0xFFF5 | $V_{spi1}$ | SPI1 |
| 4 | 0xFFF6~0xFFF7 | $V_{lol}$ | MCG 锁定丢失 |
| 3 | 0xFFF8~0xFFF9 | $V_{lvd}$ | 低电压检测 |
| 2 | 0xFFFA~0xFFFB | $V_{irq}$ | IRQ 引脚 |
| 1 | 0xFFFC~0xFFFD | $V_{swi}$ | 软件中断 |
| 0 | 0xFFFE~0xFFFF | $V_{reset}$ | 复位 |

与中断相关的寄存器只有一个，即中断引脚请求状态的控制寄存器（IRQSC）。IRQSC 寄存器用于对中断引脚的请求进行配置、报告中断状态及确认中断请求的状态和控制位。具体描述如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | 0 | IRQPDD | IRQEDG | IRQPE | IRQF | | IRQIE | IRQMOD |
| | 写 | | | | | | IRQACK | | |
| 复位 | | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

IRQPDD 是 IRQ 引脚内部上拉控制位。

IRQ 引脚激活时（IRQPE=1），该位用来控制内部上拉/下拉器件，从而允许使用外部器件。

IQRPDD=1 且 IRQPE=1，禁止 IRQ 内部上拉。

IQRPDD=0 且 IRQPE=1，允许 IRQ 内部上拉。

IRQEDG 是 IRQ 边沿选择位。

1=上升沿或上升沿/高电平有效。

0=下降沿或下降沿/低电平有效。

IRQPE 是 IRQ 引脚使能位。

1=IRQ 引脚使能。

0=禁止 IRQ 引脚。

IRQF 是 IRQ 中断标志位。

1=检测到 IRQ 中断请求。

0=无 IRQ 中断请求。

IRQACK 是中断请求确认位。

该位用来确认中断请求事件，写入 1 清除 IRQF；写入 0 则没有任何意义或影响，读总是返回 0。如果选择了边沿/电平触发（IRQMOD=1），则不能清除 IRQF。

IRQIE 是 IRQ 中断使能位。

该位决定 IRQ 事件是否生成中断请求。当 IRQIE=1 且 IRQF=1 时，产生中断请求；当 IRQIE=0 时，不产生中断请求。

IRQMOD 是 IRQ 触发模式位。

1=边沿/电平触发模式。

0=边沿触发模式。

### 3. 系统配置

JM60 在进行系统初始化时需对一些敏感器件加以配置，配置的寄存器主要有 SOPT1、SOPT2、SPMSC1、SPMSC2 等。其中 SOPT1 和 SOPT2 是 write-once 寄存器，因此只重视复位后的第一次写入，任何后续的写入尝试都无效，以避免对这些敏感器件的意外修改。它可以在任何时候读取。这两个寄存器应在用户复位初始化程序期间被写入，以设置期望的控制。SPMSC1 和 SPMSC2 包含低电压检测功能的状态和控制位。这两个寄存器应在用户复位初始化程序期间被写入，以设置期望的控制。

（1）系统选项寄存器 1（SOPT1）

SOPT1 的直观描述如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | COPT | | STOPE | | 0 | 0 | | |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

COPT[1:0]是 COP 计数器溢出周期选择位。

COPT[1:0]和 SOPT2 寄存器的 COPCLKS 位一起决定 COP 计数器的溢出周期，具体组合如表 1-2 所示。

表 1-2　COPT[1:0]和 COPCLKS 的组合表

| 控制位 | | 时钟源 | COP 窗口打开（COPW=1） | COP 溢出数 |
|---|---|---|---|---|
| COPCLKS | COPT[1:0] | | | |
| N/A | 0:0 | N/A | N/A | COP 被禁止 |
| 0 | 0:1 | 1kHz LPO 时钟 | N/A | 235 周期（32ms） |
| 0 | 1:0 | 1kHz LPO 时钟 | N/A | 28 周期（256ms) |
| 0 | 1:1 | 1kHz LPO 时钟 | N/A | 210 周期（1.024s） |
| 1 | 0:1 | BUSCLK | 6144 周期 | 213 周期 |
| 1 | 1:0 | BUSCLK | 49152 周期 | 216 周期 |
| 1 | 1:1 | BUSCLK | 196608 周期 | 218 周期 |

（2）系统选项寄存器 2（SOPT2）

SOPT2 的直观描述如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | COPCLKS | COPW | 0 | 0 | 0 | SPI1FE | SPI2FE | ACIC |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

COPCLKS 是 COP 时钟选择位。

1=COP 时钟为总线时钟。

0=COP 时钟为 1kHz。

COPW 是 COP 窗口位。

1=COP 视窗工作模式。

0=COP 正常工作。

SPI1FE 是 SPI1 端口引脚输入滤波使能位。

1=使能 SPI1 端口引脚输入滤波，以消除噪声。

0=禁止 SPI1 端口引脚输入滤波，以使能更高的 SPI 波特率。

SPI2FE 是 SPI2 端口引脚输入滤波使能位。

1=使能 SPI2 端口引脚输入滤波，以消除噪声，从而限制 SPI 的最高波特率。

0=禁止 SPI2 端口引脚输入滤波，以使能更高的 SPI 波特率。

ACIC 是模拟比较器输入捕捉使能位。

1=ACMP 输出接 TPM 输入通道 0。

0=ACMP 输出不接 TPM 输入通道 0。

（3）系统电源管理状态和控制寄存器 1（SPMSC1）

SPMC1 的直观描述如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | LVWF | 0 | LVWIE | LVDRE | LVDSE | LVDE | 0 | BGBE |
| | 写 | | LVWACK | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

LVWF 是低电压报警位。

1=低电压报警出现。

0=没有出现低电压报警。

LVWACK 是低电压报警应答位。

如果 LVWF=1，则出现低电压报警。为应答低电压报警，需置 LVWACK 位，如果此时低电压报警消失，则将自动清除 LVWF 位。

LVWIE 是低电压报警中断使能位。

1=当 LVWF=1 时产生硬件中断。

0=禁止硬件中断。

LVDRE 是低电压检测复位使能。

1=当检测到低电压事件时，强制 MCU 复位。

0=LVD 事件不产生硬件复位。

LVDSE 是低电压检测停止使能位。

1=在停止模式下使能低电压检测。

0=在停止模式下禁止低电压检测。

LVDE 是低电压检测使能位。

1=使能 LVD 功能。

0=禁止 LVD 功能。

BGBE 是带通缓冲区使能位。

1=使能带通缓冲功能。

0=禁止带通缓冲功能。

（4）系统电源管理状态和控制寄存器 2（SPMSC2）

SPMSC2 的直观描述如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | 0 | 0 | LVDV | LVWV | PPDF | 0 | 0 | PPDC |
| | 写 | | | | | | PPDACK | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

LVDV 是低电压检测电压选择位。

该位选择 LVD 动作点设置。LVDV 和 LVWV 一起决定了 LVD 和 LVDW 动作点电压，具体组合见表 1-3。

LVWV 是低电压报警电压选择位。

该位选择低电压报警动作点电压。

PPDF 是部分掉电标志位。

1=MCU 已经从 stop2 模式中恢复。

0=MCU 未从 stop2 模式中恢复。

PPDC 是部分掉电控制位。

1=stop2 模式，部分掉电使能。

0=使能 stop3 模式。

表 1-3　LVDV 和 LVWV 组合表

| LVDV:LVWV | LVW 动作点 | LVD 动作点 |
|---|---|---|
| 0:0 | $V_{LVW0}$=2.74V | $V_{LVD0}$=2.56V |
| 0:1 | $V_{LVW1}$=2.92V | |
| 1:0 | $V_{LVW2}$=4.3V | $V_{LVD1}$=4.0V |
| 1:1 | $V_{LVW3}$=4.6V | |

## 1.2.4　多功能时钟发生器（MCG）模块

JM60 的时钟源模块不再是 HCS08 系列其他单片机的内部时钟源（ICS），而是多功能时钟发生器（MCG）模块。MCG 模块的主要特性如下：

- 锁频环功能。使用内部 32kHz 参考时钟，0.2%分辨率；可以使用内部或外部参考时钟控制 FLL。
- 锁相环功能。集成压控振荡器（VCO）、模数 VCO 分频器、相位/频率检测器、环路滤波器，锁定检测器具有中断功能。
- 内部参考时钟。9 个调整位，确保精确度。可作为控制器的时钟源。
- 外部参考时钟。外部振荡器控制，具有复位功能的时钟监控器，可作为控制器的时钟源。
- 无论是 FLL 还是 PLL 模式，BDC 时钟（MCGLCLK）是一个由 DCO 输出除以 2 得出的常量。

下面将介绍 MCG 模块的组成和寄存器。

**1. MCG模块的组成**

MCG 模块为控制器提供了几个时钟源选项。MCG 模块包含一个锁频环（FLL）和一个锁相环（PLL），可以由内部或外部参考时钟控制。可以选择 FLL 或 PLL 时钟作为 MCU 系统时钟，也可以选择内部或外部参考时钟作为 MCU 系统时钟。无论选择哪个时钟源，它都要通过总线分频器，该分频器允许生成更低频率的输出时钟。MCG 还控制一个外部振荡器（XOSC），以便把晶体用做外部参考时钟。MCG 的结构如图 1-3 所示。

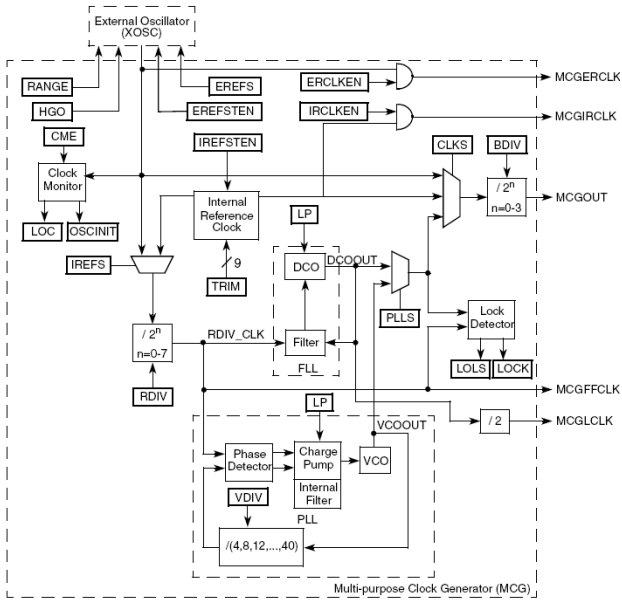



图 1-3　MCG 模块结构图

### 2. MCG 模块的寄存器

MCG 模块的寄存器主要包括：MCG 控制寄存器、MCG 修正寄存器和 MCG 状态和控制寄存器三类。下面介绍寄存器的具体功能。

（1）MCG 控制寄存器 1（MCGC1）

MCGC1 的直观描述如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | CLKS | | RDIV | | | IREFS | IRCLKEN | IREFSTEN |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

CLKS 是时钟源选择位。

00=选择 FLL 或 PLL 的输出。

01=选择内部参考时钟。

10=选择外部参考时钟。

11=预留。

RDIV 是参考分频器。

000=参考时钟。

001=参考时钟/2。

010=参考时钟/4。

011=参考时钟/8。

100=参考时钟/16。

101=参考时钟/32。

110=参考时钟/64。

111=参考时钟/128。

RDIV 对由 IREFS 位选择的参考时钟进行分频。如果选择为 FLL 输出，FLL 的参考频率必须在 31.25kHz~39.0625kHz 范围内；如果选择为 PLL 输出，PLL 的参考频率必须在 1MHz~2MHz 范围内。

IREFS 是参考时钟源选择位。

1=选择内部参考时钟。

0=选择外部参考时钟。

IRCLKEN 是内部参考时钟使能位。

1=MCGIRCLK 使能。

0=MCGIRCLK 不起作用。

该位使能内部参考时钟，用做 MCGIRCLK。

IREFSTEN 是内部参考时钟停止使能位。

1=如果 MCG 在进入停止模式前已设置了 IRCLKEN=1 或 MCG 已处于 FEI、FBI 或 BLPI 模式，那么内部参考时钟在停止模式中保持使能。

0=内部参考时钟在停止模式中禁用。

（2）MCG 控制寄存器 2（MCGC2）

MCGC2 的直观描述如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | BDIV | | RANGE | HGO | LP | EREFS | ERCLKEN | EREFSTEN |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

BDIV 是总线分频器。

00=不分频。

01=2 分频。

10=4 分频。

11=8 分频。

BDIV 对 MCGC1 寄存器中 CLKS 所选的时钟信号进行分频从而控制总线频率。

RANGE 是频率范围选择位。

1=为外部 1MHz~16MHz 振荡器选择高频率范围（外部时钟频率为 1MHz~40MHz）。

0=为外部 32kHz~100kHz 振荡器选择低频率范围（外部时钟频率为

32kHz~1MHz）。

HGO 是高增益振荡器选择位。

1=配置外部振荡器的高增益模式。

0=配置外部振荡器的低功耗模式。

LP 是低功耗选择位。

1=在旁路模式中禁止 FLL 或 PLL。

0=在旁路模式中激活 FLL 或 PLL。

EREFS 是外部参考时钟选择位。

1=选择振荡器。

0=选择外部时钟源。

ERCLKEN 是外部参考时钟使能位。

1=使能 MCGERCLK。

0=禁止 MCGERCLK。

EREFSTEN 是外部参考时钟停止使能位。

1=如果进入停止模式前已经设置了 ERCLKEN，或 MCG 已处于 FEE、FBE、PEE 或 BLPE 模式，那么外部参考时钟在停止模式中保持使能。

0=外部参考时钟在停止模式中禁用。

（3）MCG 修正寄存器（MCGTRM）

MCGTRM 寄存器通过控制内部参考时钟周期来控制内部参考时钟频率。增大 TRIM 的值将延长内部参考时钟周期从而减少其频率；反之，减少 TRIM 值会缩短内部参考时钟周期从而增加其频率。直观描述如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | TRIM | | | | | | | |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

（4）MCG 状态和控制寄存器（MCGSC）

直观描述如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | LOLS | LOCK | PLLST | IREFST | CLKST | | OSCINIT | FTRIM |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

LOLS 是锁定状态丢失位。

1=FLL 或 PLL 锁定丢失。

0=FLL 或 PLL 锁定未丢失。

LOCK 是锁定状态。

1=FLL 或 PLL 已锁定。

0=FLL 或 PLL 未锁定。

PLLST 是 PLL 状态选择位。

1=PLLS 的时钟源为 PLL 时钟。

0=PLLS 的时钟源为 FLL 时钟。

IREFST 是内部参考状态位。

1=参考时钟源是内部参考时钟。

0=参考时钟源是外部参考时钟。

由于内部时钟域间的同步，在向 IREFS 进行写入后，IREFST 不会立即被更新。

CLKST 是当前时钟模式。

00：选择 FLL 输出。

01：选择内部参考时钟。

10：选择外部参考时钟。

11：选择 PLL 输出。

由于内部时钟域间的同步，在向 CLKS 进行写入后，CLKST 不会立即被更新。

OSCINIT 是 OSC 初始化。

如果 ERCLKEN=1，即外部参考时钟使能，或者 MCG 正处于 FEE、FBE、PEE 或 BLPE 模式，并且设置了 EREFS=1，那么在完成了外部振荡器时钟的初始化周期后就要置位该位。只有当 EREFS 被清零或 MCG 处于 FEI、FBI 或 BLPI 模式之一，且 ERCLKEN 被清零时，这个位才被清零。

FTRIM 是 MCG 微调。

控制内部参考时钟频率最细微的调节。FTRIM=1，会以最小的幅度延长该时段；FTRIM=0，会以最小的幅度缩短该时段。

（5）MCG 控制寄存器 3（MCGC3）

MCGC3 直观描述如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | LOLIE | PLLS | CME | 0 | VDIV | | | |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

LOLIE 是锁定丢失中断允许位。

1=允许锁定丢失中断。

0=禁止锁定丢失中断。

PLLS 是 PLL 选择位。

1=选择 PLL。

0=选择 FLL。

CME 是时钟监视器允许位。

1=允许外部时钟丢失中断。

0=禁止时钟监视器。

VDIV 是 VCO 分频因子位。

0000：预留。

0001：4 分频。

0010：8 分频。

0011：12 分频。

0100：16 分频。

0101：20 分频。

0110：24 分频。

0111：28 分频。

1000：32 分频。

1001：36 分频。

1010：40 分频。

1011：预留

11xx：预留。

## 1.2.5　I/O 端口

I/O（Input/Output）即基本的输入/输出，有时也称为并行 I/O。作为普通输入引脚，程序可以读取该引脚，对应引脚是 1（高电平）或 0（低电平），即开关量输入。作为普通输出引脚，单片机内部程序向该引脚输出 1（高电平）或 0（低电平），即开关量输出。MC9S08 单片机有多个普通 I/O 口，如 A 口和 B 口，作为普通 I/O 口，它们的每一个引脚均可通过相应口的“数据方向寄存器”独立地设置为输入或输出。在实际电路中，对于未使用的 I/O 口引脚，应该将其接 VDD 或 VSS，不要悬空，以减少干扰。

### 1.2.5.1　A 口

A 口的 6 只引脚仅作为普通输入/输出并行口使用，不具有其他功能。

A 口作为普通 I/O 口时，具有 5 个寄存器：A 口数据寄存器（PTAD）、A 口数据方向寄存器（PTADD）、A 口上拉电阻允许寄存器（PTAPE）、A 口输出转换率控制寄存器

（PTASE）和 A 口输出驱动能力选择寄存器（PTADS）。

**1. A口数据寄存器（PTAD）**

A 口数据寄存器的地址是$0000，其 D5~D0 位分别记为 PTAD5～PTAD0。若 A 口的某一引脚 PTAD*x* 被定义成输出，程序使 A 口数据寄存器 PTAD 的相应位 PTAD*x*=0，则引脚 PTAD*x* 输出低电平；程序使 PTAD*x*=1，则引脚 PTAD*x* 输出高电平。若 A 口的某一引脚 PTAD*x* 被定义成输入，程序通过读取 A 口数据寄存器 PTAD 以获得输入情况：0 表示输入为低电平；1 表示输入为高电平。

A 口数据寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | PTAD5 | PTAD4 | PTAD3 | PTAD2 | PTAD1 | PTAD0 |

其中 D5~D0（PTAD5~PTAD0）为 A 口数据位。

这些读/写位可软件编程。每一个引脚的方向由相应的数据方向寄存器控制（PTAD2 只能作为输入）。复位对数据寄存器没有影响。

**2. A口数据方向寄存器（PTADD，Data Direction Register A）**

A 口数据方向寄存器的地址是$0001，其 D5~D0 位分别记为 PTADD5~PTADD0。这些位分别控制着 A 口引脚 PTAD5～PTAD0 是输入还是输出。复位时 PTADD 为$00。

A 口数据方向寄存器位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | PTADD5 | PTADD4 | PTADD3 | PTADD2 | PTADD1 | PTADD0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

若 PTADD*x*=0，则引脚 PTA*x* 为输入；若 PTADD*x*=1，则引脚 PTA*x* 为输出。

**3. A口上拉电阻允许寄存器（PTAPE，Port A Input Pullup Enable Register）**

A 口上拉电阻允许寄存器的地址是$1840，其 D5~D0 位分别记为 PTAPE5~PTAPE0。

A 口上拉电阻允许寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | PTAPE5 | PTAPE4 | PTAPE3 | PTAPE2 | PTAPE1 | PTAPE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D5~D0（PTAPE5~PTAPE0）为 A 口输入上拉允许位。

PTAPE*x*=1，代表 PTA 相应位设置为 0 并且没有设置其他复用功能时，相应 A 口引脚有内部上拉电阻。

**提示**

在引脚被定义成输入时，可通过上拉电阻允许寄存器定义有无内部上拉电阻：0 表示没有内部上拉电阻；1 表示有内部上拉电阻。

**4. A口输出转换率控制寄存器（PTASE）**

A 口输出转换率控制寄存器的地址是$1841，其 D5~D0 位分别记为 PTASE5~PTASE0。

A 口输出转换率控制寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | PTASE5 | PTASE4 | PTASE3 | PTASE2 | PTASE1 | PTASE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D5~D0（PTASE5~PTASE0）为 A 口输出转换率允许位。

PTASE*x*=1，代表 PTA 相应位设置为 0 并且没有设置其他复用功能时，相应 A 口引脚输出转换率允许。

**5. A口输出驱动能力选择寄存器（PTADS）**

A 口输出驱动能力选择寄存器的地址是$1842，其 D5~D0 位分别记为 PTADS5~PTADS0。

A 口输出驱动能力选择寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | PTADS5 | PTADS4 | PTADS3 | PTADS2 | PTADS1 | PTADS0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D5~D0（PTADS5~PTADS0）为 A 口输出驱动能力选择位。

PTADS*x*=1，代表 PTA 相应位设置为 0 并且没有设置其他复用功能时，相应 A 口引脚的高输出驱动能力。

### 1.2.5.2　B 口

B 口是 8 位输入、输出并行口，与键盘中断（KBI）、ADC 输入通道等复用。

B 口作为普通 I/O 口时，具有 5 个寄存器：B 口数据寄存器（PTBD）、B 口数据方向寄存器(PTBDD)、B 口上拉电阻允许寄存器(PTBPE)、B 口输出转换率控制寄存器(PTBSE)和 B 口输出驱动能力选择寄存器（PTBDS）。

**1. B口数据寄存器（PTBD）**

B 口数据寄存器的地址是$0002，其 D7~D0 位分别记为 PTBD7～PTAD0。若 B 口的某一引脚 PTBD*x* 被定义成输出，程序使 B 口数据寄存器 PTBD 的相应位 PTBD*x*=0，则引脚 PTBD*x* 输出低电平；程序使 PTBD*x*=1，则引脚 PTBD*x* 输出高电平。若 B 口的某一引脚 PTBD*x* 被定义成输入，程序通过读取 B 口数据寄存器 PTBD，以获得输入情况：0 表示输入为低电平；1 表示输入为高电平。

B 口数据寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTBD7 | PTBD6 | PTBD5 | PTBD4 | PTBD3 | PTBD2 | PTBD1 | PTBD0 |

D7~D0（PTBD7~PTBD0）为 B 口数据位。

这些读/写位可软件编程。每一个引脚的方向由相应的数据方向寄存器控制。复位对数据寄存器没有影响。

**2. B口数据方向寄存器（PTBDD，Data Direction Register B）**

B 口数据方向寄存器的地址是$0003，其 D7~D0 位分别记为 PTBDD7~PTBDD0。这些位分别控制着 B 口引脚 PTBD7～PTBD0 是输入还是输出。复位时 PTBDD 为$00。

B 口数据方向寄存器位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTBDD7 | PTBDD6 | PTBDD5 | PTBDD4 | PTBDD3 | PTBDD2 | PTBDD1 | PTBDD0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

若 PTBDD*x*=0，则引脚 PTB*x* 为输入；若 PTBDD*x*=1，则引脚 PTBx 为输出。

**3. B口上拉电阻允许寄存器（PTBPE，Port B Input Pullup Enable Register）**

B 口上拉电阻允许寄存器的地址是$1844，其 D7~D0 位分别记为 PTBPE7~PTBPE0。

B 口上拉电阻允许寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTBPE7 | PTBPE6 | PTBPE5 | PTBPE4 | PTBPE3 | PTBPE2 | PTBPE1 | PTBPE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D5~D0（PTBPE7~PTBPE0）为 B 口输入上拉允许位。

PTBPE*x*=1，代表 PTB 相应位设置为 0 并且没有设置其他复用功能时，相应 B 口引脚有内部上拉电阻。

**提示**

在引脚被定义成输入时，可通过上拉电阻允许寄存器定义有无内部上拉电阻：0 表示没有内部上拉电阻；1 表示有内部上拉电阻。

**4. B口输出转换率控制寄存器（PTBSE）**

B 口输出转换率控制寄存器的地址是$1845，其 D7~D0 位分别记为 PTBSE7~PTBSE0。

B 口输出转换率控制寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTBSE7 | PTBSE6 | PTBSE5 | PTBSE4 | PTBSE3 | PTBSE2 | PTBSE1 | PTBSE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D7~D0（PTBSE7~PTBSE0）为 B 口输出转换率允许位。

PTBSE*x*=1，代表 PTB 相应位设置为 0 并且没有设置其他复用功能时，相应 B 口引脚输出转换率允许。

**5. B口输出驱动能力选择寄存器（PTBDS）**

B 口输出驱动能力选择寄存器的地址是$1846，其 D7~D0 位分别记为 PTBDS7~PTBDS0。

B 口输出驱动能力选择寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTBDS7 | PTBDS6 | PTBDS5 | PTBDS4 | PTBDS3 | PTBDS2 | PTBDS1 | PTBDS0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D7~D0（PTBDS5~PTBDS0）为 B 口输出驱动能力选择位。

PTBDS*x*=1，代表 PTB 相应位设置为 0 并且没有设置其他复用功能时，相应 B 口引脚的高输出驱动能力。

### 1.2.5.3　C 口

C 口是 7 位输入、输出并行口，与串行通信接口（SCI2）、IIC 等复用。

C 口作为普通 I/O 口时，具有 5 个寄存器：C 口数据寄存器（PTCD）、C 口数据方向寄存器(PTCDD)、C 口上拉电阻允许寄存器(PTCPE)、C 口输出转换率控制寄存器(PTCSE)和 C 口输出驱动能力选择寄存器（PTCDS）。

**1. C口数据寄存器（PTCD）**

C 口数据寄存器的地址是$0004，其 D6~D0 位分别记为 PTCD6～PTCD0。若 C 口的某一引脚 PTCD*x* 被定义成输出，程序使 C 口数据寄存器 PTCD 的相应位 PTCD*x*=0，则引脚 PTCD*x* 输出低电平；程序使 PTCD*x*=1，则引脚 PTCD*x* 输出高电平。若 C 口的某一引脚 PTCD*x* 被定义成输入，程序通过读取 C 口数据寄存器 PTCD 以获得输入情况：0 表示输入为低电平；1 表示输入为高电平。

C 口数据寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | PTCD6 | PTCD5 | PTCD4 | PTCD3 | PTCD2 | PTCD1 | PTCD0 |

D6~D0（PTCD6~PTCD0）为 C 口数据位。

这些读/写位可软件编程。每一个引脚的方向由相应的数据方向寄存器控制。复位对数据寄存器没有影响。

**2. C口数据方向寄存器（PTCDD，Data Direction Register C）**

C 口数据方向寄存器的地址是$0005，其 D6~D0 位分别记为 PTCDD6~PTCDD0。这些位分别控制着 C 口引脚 PTCD6～PTCD0 是输入还是输出。复位时 PTCDD 为$00。

C 口数据方向寄存器位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | PTCDD6 | PTCDD5 | PTCDD4 | PTCDD3 | PTCDD2 | PTCDD1 | PTCDD0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

若 PTCDD*x*=0，则引脚 PTC*x* 为输入；若 PTCDD*x*=1，则引脚 PTC*x* 为输出。

**3. C口上拉电阻允许寄存器（PTCPE，Port C Input Pullup Enable Register）**

C 口上拉电阻允许寄存器的地址是$1848，其 D6~D0 位分别记为 PTCPE6~PTCPE0。

C 口上拉电阻允许寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | PTCPE6 | PTCPE5 | PTCPE4 | PTCPE3 | PTCPE2 | PTCPE1 | PTCPE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D6~D0（PTCPE6~PTCPE0）为 C 口输入上拉允许位。

PTCPE*x*=1，代表 PTC 相应位设置为 0 并且没有设置其他复用功能时，相应 C 口引脚有内部上拉电阻。

**提示**

在引脚被定义成输入时，可通过上拉电阻允许寄存器定义有无内部上拉电阻：0 表示没有内部上拉电阻；1 表示有内部上拉电阻。

**4. C口输出转换率控制寄存器（PTCSE）**

C 口输出转换率控制寄存器的地址是$1849，其 D6~D0 位分别记为 PTCSE6~PTCSE0。

C 口输出转换率控制寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | PTCSE6 | PTCSE5 | PTCSE4 | PTCSE3 | PTCSE2 | PTCSE1 | PTCSE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D6~D0（PTCSE6~PTCSE0）为 C 口输出转换率允许位。

PTCSE*x*=1，代表 PTC 相应位设置为 0 并且没有设置其他复用功能时，相应 C 口引脚输出转换率允许。

**5. C口输出驱动能力选择寄存器（PTCDS）**

C 口输出驱动能力选择寄存器的地址是$184A，其 D6~D0 位分别记为 PTCDS6~PTCDS0。

C 口输出驱动能力选择寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | PTCDS6 | PTCDS5 | PTCDS4 | PTCDS3 | PTCDS2 | PTCDS1 | PTCDS0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D6~D0（PTCDS6~PTCDS0）为 C 口输出驱动能力选择位。

PTCDS*x*=1，代表 PTC 相应位设置为 0 并且没有设置其他复用功能时，相应 C 口引脚的高输出驱动能力。

#### 1.2.5.4　D 口

D 口是 8 位输入、输出并行口，与键盘中断（KBI）、模拟比较器、ADC 输入通道等复用。

D 口作为普通 I/O 口时，具有 5 个寄存器：D 口数据寄存器（PTDD）、D 口数据方向寄存器（PTDDD）、D 口上拉电阻允许寄存器（PTDPE）、D 口输出转换率控制寄存器（PTDSE）和 D 口输出驱动能力选择寄存器（PTDDS）。

**1. D口数据寄存器（PTDD）**

D 口数据寄存器的地址是$0006，其 D7~D0 位分别记为 PTDD7～PTDD0。若 D 口的某一引脚 PTDD*x* 被定义成输出，程序使 D 口数据寄存器 PTDD 的相应位 PTDD*x*=0，则引脚 PTDD*x* 输出低电平；程序使 PTDD*x*=1，则引脚 PTDD*x* 输出高电平。若 D 口的某一引脚 PTDD*x* 被定义成输入，程序通过读取 D 口数据寄存器 PTDD 以获得输入情况：0 表示输入为低电平；1 表示输入为高电平。

D 口数据寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTDD7 | PTDD6 | PTDD5 | PTDD4 | PTDD3 | PTDD2 | PTDD1 | PTDD0 |

D7~D0（PTDD7~PTDD0）为 D 口数据位。

这些读/写位可软件编程。每一个引脚的方向由相应的数据方向寄存器控制。复位对数据寄存器没有影响。

**2. D口数据方向寄存器（PTDDD，Data Direction Register D）**

D 口数据方向寄存器的地址是$0007，其 D7~D0 位分别记为 PTDDD7~PTDDD0。这些位分别控制着 D 口引脚 PTD7～PTD0 是输入还是输出。复位时 PTDDD 为$00。

D 口数据方向寄存器位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTDDD7 | PTDDD6 | PTDDD5 | PTDDD4 | PTDDD3 | PTDDD2 | PTDDD1 | PTDDD0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

若 PTDDD*x*=0，则引脚 PTD*x* 为输入；若 PTDDD*x*=1，则引脚 PTD*x* 为输出。

**3. D口上拉电阻允许寄存器（PTDPE，Port D Input Pullup Enable Register）**

D 口上拉电阻允许寄存器的地址是$184C，其 D7~D0 位分别记为 PTDPE7~PTDPE0。

D 口上拉电阻允许寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTDPE7 | PTDPE6 | PTDPE5 | PTDPE4 | PTDPE3 | PTDPE2 | PTDPE1 | PTDPE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D7~D0（PTDPE7~PTDPE0）为 D 口输入上拉允许位。

PTDPE*x*=1，代表 PTD 相应位设置为 0 并且没有设置其他复用功能时，相

应 D 口引脚有内部上拉电阻。

**提示：**

在引脚被定义成输入时，可通过上拉电阻允许寄存器定义有无内部上拉电阻：0 表示没有内部上拉电阻；1 表示有内部上拉电阻。

**4. D口输出转换率控制寄存器（PTDSE）**

D 口输出转换率控制寄存器的地址是$184D，其 D7~D0 位分别记为 PTDSE7~PTDSE0。

D 口输出转换率控制寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTDSE7 | PTDSE6 | PTDSE5 | PTDSE4 | PTDSE3 | PTDSE2 | PTDSE1 | PTDSE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D7~D0（PTDSE7~PTDSE0）为 D 口输出转换率允许位。

PTDSE*x*=1，代表 PTD 相应位设置为 0 并且没有设置其他复用功能时，相应 D 口引脚输出转换率允许。

**5. D口输出驱动能力选择寄存器（PTDDS）**

D 口输出驱动能力选择寄存器的地址是$184E，其 D7~D0 位分别记为 PTDDS7~PTDDS0。

D 口输出驱动能力选择寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTDDS7 | PTDDS6 | PTDDS5 | PTDDS4 | PTDDS3 | PTDDS2 | PTDDS1 | PTDDS0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D7~D0（PTDDS7~PTDDS0）为 D 口输出驱动能力选择位。

PTDDS*x*=1，代表 PTD 相应位设置为 0 并且没有设置其他复用功能时，相应 D 口引脚的高输出驱动能力。

### 1.2.5.5 E 口

E 口是 8 位输入、输出并行口，与定时器/脉宽调制器（PWM）、串行通信接口（SCI1）、串行外设接口（SPI1）等复用。

E 口作为普通 I/O 口时，具有 5 个寄存器：E 口数据寄存器（PTED）、E 口数据方向寄存器(PTEDD)、E 口上拉电阻允许寄存器(PTEPE)、E 口输出转换率控制寄存器(PTESE)和 E 口输出驱动能力选择寄存器（PTEDS）。

**1. E口数据寄存器（PTED）**

E 口数据寄存器的地址是$0008，其 D7~D0 位分别记为 PTED7～PTED0。若 E 口的某一引脚 PTED*x* 被定义成输出，程序使 E 口数据寄存器 PTED 的相应位 PTED*x*=0，则引脚

PTED*x*输出低电平;程序使PTED*x*=1,则引脚PTED*x*输出高电平。若E口的某一引脚PTED*x*被定义成输入，程序通过读取 E 口数据寄存器 PTED 以获得输入情况：0 表示输入为低电平；1 表示输入为高电平。

E 口数据寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTED7 | PTED6 | PTED5 | PTED4 | PTED3 | PTED2 | PTED1 | PTED0 |

D7~D0（PTED7~PTED0）为 E 口数据位。

这些读/写位可软件编程。每一个引脚的方向由相应的数据方向寄存器控制。复位对数据寄存器没有影响。

**2. E口数据方向寄存器（PTEDD，Data Direction Register E）**

E 口数据方向寄存器的地址是$0009，其 D7~D0 位分别记为 PTEDD7~PTEDD0。这些位分别控制着 E 口引脚 PTED7～PTED0 是输入还是输出。复位时 PTEDD 为$00。

E 口数据方向寄存器位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTEDD7 | PTEDD6 | PTEDD5 | PTEDD4 | PTEDD3 | PTEDD2 | PTEDD1 | PTEDD0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

若 PTEDD*x*=0，则引脚 PTE*x* 为输入；若 PTEDD*x*=1，则引脚 PTE*x* 为输出。

**3. E口上拉电阻允许寄存器（PTEPE，Port E Input Pullup Enable Register）**

E 口上拉电阻允许寄存器的地址是$1850，其 D7~D0 位分别记为 PTEPE7~PTEPE0。

E 口上拉电阻允许寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTEPE7 | PTEPE6 | PTEPE5 | PTEPE4 | PTEPE3 | PTEPE2 | PTEPE1 | PTEPE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D7~D0（PTEPE7~PTEPE0）为 E 口输入上拉允许位。

PTEPE*x*=1，代表 PTE 相应位设置为 0 并且没有设置其他复用功能时，相应 E 口引脚有内部上拉电阻。

**提示：**

在引脚被定义成输入时，可通过上拉电阻允许寄存器定义有无内部上拉电阻：0 表示没有内部上拉电阻；1 表示有内部上拉电阻。

**4. E口输出转换率控制寄存器（PTESE）**

E 口输出转换率控制寄存器的地址是$1851，其 D7~D0 位分别记为 PTESE7~PTESE0。

E 口输出转换率控制寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTESE7 | PTESE6 | PTESE5 | PTESE4 | PTESE3 | PTESE2 | PTESE1 | PTESE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D7~D0（PTESE7~PTESE0）为 E 口输出转换率允许位。

PTESE*x*=1，代表 PTE 相应位设置为 0 并且没有设置其他复用功能时，相应 E 口引脚输出转换率允许。

**5. E口输出驱动能力选择寄存器（PTEDS）**

E 口输出驱动能力选择寄存器的地址是$1852，其 D7~D0 位分别记为 PTEDS7~PTEDS0。

E 口输出驱动能力选择寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTEDS7 | PTEDS6 | PTEDS5 | PTEDS4 | PTEDS3 | PTEDS2 | PTEDS1 | PTEDS0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D7~D0（PTEDS7~PTEDS0）为 E 口输出驱动能力选择位。

PTEDS*x*=1，代表 PTE 相应位设置为 0 并且没有设置其他复用功能时，相应 E 口引脚的高输出驱动能力。

### 1.2.5.6 F 口

F 口是 8 位输入、输出并行口，与定时器/脉宽调制器（PWM）通道等复用。

F 口作为普通 I/O 口时，具有 5 个寄存器：F 口数据寄存器（PTFD）、F 口数据方向寄存器（PTFDD）、F 口上拉电阻允许寄存器（PTFPE）、F 口输出转换率控制寄存器（PTFSE）和 F 口输出驱动能力选择寄存器（PTFDS）。

**1. F口数据寄存器（PTFD）**

F 口数据寄存器的地址是$000A，其 D7~D0 位分别记为 PTFD7～PTFD0。若 F 口的某一引脚 PTFD*x* 被定义成输出，程序使 F 口数据寄存器 PTFD 的相应位 PTFD*x*=0，则引脚 PTFD*x* 输出低电平；程序使 PTFD*x*=1，则引脚 PTFD*x* 输出高电平。若 F 口的某一引脚 PTFD*x* 被定义成输入，程序通过读取 F 口数据寄存器 PTFD 以获得输入情况，0 表示输入为低电平；1 表示输入为高电平。

F 口数据寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTFD7 | PTFD6 | PTFD5 | PTFD4 | PTFD3 | PTFD2 | PTFD1 | PTFD0 |

D7~D0（PTFD7~PTFD0）为 F 口数据位。

这些读/写位可软件编程。每一个引脚的方向由相应的数据方向寄存器控制。复位对数据寄存器没有影响。

**2. F口数据方向寄存器（PTFDD，Data Direction Register F）**

F 口数据方向寄存器的地址是$000B，其 D7~D0 位分别记为 PTFDD7~PTFDD0。这些位分别控制着 F 口引脚 PTF7～PTF0 是输入还是输出。复位时 PTFDD 为$00。

F 口数据方向寄存器位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTFDD7 | PTFDD6 | PTFDD5 | PTFDD4 | PTFDD3 | PTFDD2 | PTFDD1 | PTFDD0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

若 PTFDD*x*=0，则引脚 PTF*x* 为输入；若 PTFDD*x*=1，则引脚 PTF*x* 为输出。

**3. F口上拉电阻允许寄存器（PTFPE，Port F Input Pullup Enable Register）**

F 口上拉电阻允许寄存器的地址是$1854，其 D7~D0 位分别记为 PTFPE7~PTFPE0。

F 口上拉电阻允许寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTFPE7 | PTFPE6 | PTFPE5 | PTFPE4 | PTFPE3 | PTFPE2 | PTFPE1 | PTFPE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D7~D0（PTFPE7~PTFPE0）为 F 口输入上拉允许位。

PTFPE*x*=1，代表 PTF 相应位设置为 0 并且没有设置其他复用功能时，相应 F 口引脚有内部上拉电阻。

**提示：**

在引脚被定义成输入时，可通过上拉电阻允许寄存器定义有无内部上拉电阻：0 表示没有内部上拉电阻；1 表示有内部上拉电阻。

**4. F口输出转换率控制寄存器（PTFSE）**

F 口输出转换率控制寄存器的地址是$1855，其 D7~D0 位分别记为 PTFSE7~PTFSE0。

F 口输出转换率控制寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTFSE7 | PTFSE6 | PTFSE5 | PTFSE4 | PTFSE3 | PTFSE2 | PTFSE1 | PTFSE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D7~D0（PTFSE7~PTFSE0）为 F 口输出转换率允许位。

PTFSE*x*=1，代表 PTF 相应位设置为 0 并且没有设置其他复用功能时，相应 F 口引脚输出转换率允许。

**5. F口输出驱动能力选择寄存器（PTFDS）**

F 口输出驱动能力选择寄存器的地址是$1856，其 D7~D0 位分别记为 PTFDS7~PTFDS0。

F 口输出驱动能力选择寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | PTFDS7 | PTFDS6 | PTFDS5 | PTFDS4 | PTFDS3 | PTFDS2 | PTFDS1 | PTFDS0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D7~D0（PTFDS5~PTFDS0）为 F 口输出驱动能力选择位。

PTFDS*x*=1，代表 PTF 相应位设置为 0 并且没有设置其他复用功能时，相应 F 口引脚的高输出驱动能力。

#### 1.2.5.7 G 口

G 口是 6 位输入、输出并行口，与键盘中断（KBI）、晶振引脚等复用。

G 口作为普通 I/O 口时，具有 5 个寄存器：G 口数据寄存器（PTGD）、G 口数据方向寄存器（PTGDD）、G 口上拉电阻允许寄存器（PTGPE）、G 口输出转换率控制寄存器（PTGSE）和 G 口输出驱动能力选择寄存器（PTGDS）。

**1. G口数据寄存器（PTGD）**

G 口数据寄存器的地址是$000C，其 D5~D0 位分别记为 PTGD5～PTGD0。若 G 口的某一引脚 PTGD*x* 被定义成输出，程序使 G 口数据寄存器 PTGD 的相应位 PTGD*x*=0，则引脚 PTGD*x* 输出低电平；程序使 PTGD*x*=1，则引脚 PTGD*x* 输出高电平。若 G 口的某一引脚 PTGD*x* 被定义成输入，程序通过读取 G 口数据寄存器 PTGD 以获得输入情况，0 表示输入为低电平；1 表示输入为高电平。

G 口数据寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | PTGD5 | PTGD4 | PTGD3 | PTGD2 | PTGD1 | PTGD0 |

D5~D0（PTGD5~PTGD0）为 G 口数据位。

这些读/写位可软件编程。每一个引脚的方向由相应的数据方向寄存器控制。复位对数据寄存器没有影响。

**2. G口数据方向寄存器（PTGDD，Data Direction Register G）**

G 口数据方向寄存器的地址是$000D，其 D5~D0 位分别记为 PTGDD5~PTGDD0。这些位分别控制着 G 口引脚 PTGD5～PTGD0 是输入还是输出。复位时 PTGDD 为$00。

G 口数据方向寄存器位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | PTGDD5 | PTGDD4 | PTGDD3 | PTGDD2 | PTGDD1 | PTGDD0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

若 PTGDD*x*=0，则引脚 PTG*x* 为输入；若 PTGDD*x*=1，则引脚 PTG*x* 为输出。

**3. G口上拉电阻允许寄存器（PTGPE，Port G Input Pullup Enable Register）**

G 口上拉电阻允许寄存器的地址是$1858，其 D5~D0 位分别记为 PTGPE5~PTGPE0。

G 口上拉电阻允许寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | PTGPE5 | PTGPE4 | PTGPE3 | PTGPE2 | PTGPE1 | PTGPE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D5~D0（PTGPE5~PTGPE0）为 G 口输入上拉允许位。

PTGPE*x*=1，代表 PTG 相应位设置为 0 并且没有设置其他复用功能时，相应 G 口引脚有内部上拉电阻。

**提示：**

在引脚被定义成输入时，可通过上拉电阻允许寄存器定义有无内部上拉电阻：0 表示没有内部上拉电阻；1 表示有内部上拉电阻。

**4. G口输出转换率控制寄存器（PTGSE）**

G 口输出转换率控制寄存器的地址是$1859，其 D5~D0 位分别记为 PTGSE5~PTGSE0。G 口输出转换率控制寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | PTGSE5 | PTGSE4 | PTGSE3 | PTGSE2 | PTGSE1 | PTGSE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D5~D0（PTGSE7~PTGSE0）为 G 口输出转换率允许位。

PTGSE*x*=1，代表 PTG 相应位设置为 0 并且没有设置其他复用功能时，相应 G 口引脚输出转换率允许。

**5. G口输出驱动能力选择寄存器（PTGDS）**

G 口输出驱动能力选择寄存器的地址是$185A，其 D5~D0 位分别记为 PTGDS7~PTGDS0。G 口输出驱动能力选择寄存器的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | PTGDS5 | PTGDS4 | PTGDS3 | PTGDS2 | PTGDS1 | PTGDS0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

D5~D0（PTGDS5~PTGDS0）为 G 口输出驱动能力选择位。

PTGDS*x*=1，代表 PTG 相应位设置为 0 并且没有设置其他复用功能时，相应 G 口引脚的高输出驱动能力。

## 1.2.6　KBI 模块

MC9S08JM60 单片机 B 口的 PTB4 和 PTB5，D 口的 PTD2、PTD3，G 口的 PTG0～PTG3，共 8 只引脚与键盘中断模块 KBI（Keybord Interrupt Module）的引脚复用。

键盘中断模块的主要特性如下：

- 具有 8 个独立的可屏蔽的外部键盘中断，每个键盘中断有键盘中断允许位。

- 具有 1 个自动唤醒逻辑控制的内部中断，有独立键盘中断允许位，共享键盘中断屏蔽位。
- 可以选择下降沿触发或者下降沿和低电平触发。
- 作为键盘中断引脚输入时内部电阻上拉。
- 可从低功耗模式中唤醒。

**1. 键盘状态和控制寄存器KBISC**

KBISC 包含了键盘中断请求标志、屏蔽键盘中断请求、定义键盘中断的触发方式和消除键盘中断标志等功能。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | 0 | 0 | KBF | KBACK | KBIE | KBMOD |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

KBF（只读位）为键盘中断标志位（Keybord Interrupt Flag Bit）。

KBF=1，键盘中断已产生；

KBF=0，键盘中断未产生。

KBACK（只写位）为键盘中断请求确认位（Keybord Interrupt Acknowledge Bit）。

向此位写入 1，则清除键盘中断请求和自动唤醒逻辑，ACKK 读取值总为 0。

KBIE 为键盘中断屏蔽位（Keybord Interrupt Mask Bit）。

KBIE=0，屏蔽键盘中断请求；KBIE=1，开放键盘中断。

KBMOD 为键盘中断触发方式选择位。

KBMOD=1，边沿及电平触发；KBMOD=0，边沿触发。

**2. 键盘中断允许寄存器KBIPE**

KBIPE 的各位决定所对应的引脚是否允许中断进入。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | KBIPE7 | KBIPE6 | KBIPE5 | KBIPE4 | KBIPE3 | KBIPE2 | KBIPE1 | KBIPE0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

KBIPEn 为键盘中断允许位。

若 KBIPE*x*=1，表示相应引脚 KBIEP*x*（也就是引脚 PTA*x*）被定义为中断引入引脚。复位时，全为 0。

**3. 键盘边沿选择寄存器KBIES**

KBIES 的各位决定所对应的引脚的边沿/电平触发方式。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | KBEDG7 | KBEDG6 | KBEDG5 | KBEDG4 | KBEDG3 | KBEDG2 | KBEDG1 | KBEDG0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

KBEDGn 为键盘中断边沿/电平选择位。若 KBEDG*x*=1，表示相应引脚被定义为下降沿/高电平中断输入；若 KBEDG*x*=0，表示相应引脚被定义为上升沿/低电平中断输入；复位时，全为 0。

### 1.2.7　定时器/PWM 模块

定时器在单片机系统中是非常重要的部分，它在事件控制与管理方面有着重要的应用。MC9S08JM60 内部包括两个独立的定时器。

MC9S08JM60 定时器的功能结构如图 1-4 所示。每个定时器具有两路独立的输入捕捉和输出比较通道；可以实现带缓冲和不带缓冲的脉宽调制波；可以得到 8 种可编程选择的分频因子；可以自由运行计数或预置计数；可以停止或清除计数器计数；定时器溢出可以引发中断，每路通道的输入捕捉和输出比较也可以引发中断。

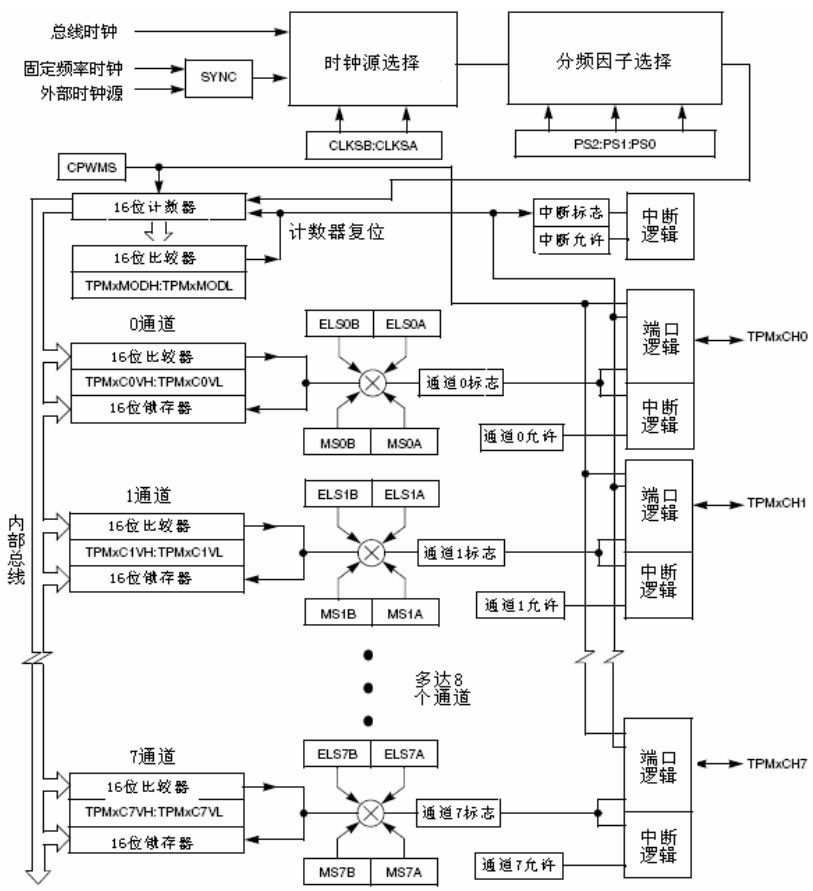



图 1-4　定时器功能结构图

每一路定时器都由以下寄存器控制。

**1. 定时器状态和控制寄存器TPMxSC**

TPMxSC 包含了定时器中断标志、中断请求、时钟源选择和分频因子等功能。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | TOF | TOIE | CPWMS | CLKSB | CLKSA | PS2 | PS1 | PS0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

TOF 为定时器溢出标志位。该位用来标志定时器的溢出。当定时器的计数达到$FFFF（自由计数运行状态）或预置计数器的值时，定时器的计数将清零，同时该位置为 1。读取定时器状态和控制寄存器并向该位写入 0，可以清除该标志位。

该位虽然是可读/写的，但由于该位只是状态位，向该位写入 1 的操作仅仅完成清除该标志位的作用，没有任何实际意义。另外需要注意，如果清除该标志位前又发生了定时器溢出，那么这一次清零将无法完成。程序必须保证每一次溢出发生后，都要在下一次溢出发生之前清除该标志位。在使用溢出中断的情况下，要求在中断返回之前必须清除溢出标志位，并且该中断的处理时间在最不利的情形下也不能超过一次计数溢出时间。程序在向该位写入 1 进行清除后，可以再次读取该位，判断是否已经正确清零，这样就能够发现连续的定时器溢出的情况。这种情况往往是因为程序没能及时地处理定时器溢出事件。

TOIE 为定时器溢出中断允许位。该位用来标志是否允许定时器溢出中断。该位为 1，表示允许溢出中断，当发生溢出时，系统会执行溢出中断处理程序；该位为 0，表示不允许溢出中断。该位是可读/写的，读取的数值就是其实际值。

CPWMS 为 PWM 中心对齐控制位。该位控制 PWM 操作模式。该位为 1，则所有的定时器通道都工作在中心对齐的脉宽调制模式下；该位为 0，则每个定时器通道分别由各自的控制寄存器控制作为输入捕捉、输出比较或者边界对齐的脉宽调制。

CLKSB：CLKSA 为时钟源选择控制位。表 1-4 列举了 CLKSB：CLKSA 的值与时钟源选择的关系。

**表 1-4　定时器时钟源选择**

| CLKSB：CLKSA | 时钟源 |
|---|---|
| 0：0 | 没有时钟，定时器禁止 |
| 0：1 | 总线时钟（BUSCLK） |
| 1：0 | 固定的系统时钟（XCLK） |
| 1：1 | 外部时钟（TPMx Ext Clk） |

PS2：PS1：PS0 为定时器分频因子选择控制位。

这 3 位共同设定定时器的分频因子。3 位的不同组合有 8 种可能。表 1-5 列举了 PS2：

PS1：PS0 的值与分频因子的关系。

表 1-5　选择分频因子

| PS2：PS1：PS0 | 分频因子 |
|---|---|
| 0：0：0 | 1 |
| 0：0：1 | 2 |
| 0：1：0 | 4 |
| 0：1：1 | 8 |
| 1：0：0 | 16 |
| 1：0：1 | 32 |
| 1：1：0 | 64 |
| 1：1：1 | 128 |

定时器的状态和控制寄存器的各位都有不同的作用，在设定时可以只对需要更改的某一位进行读/写，也可以对整个寄存器进行读/写。

**2. 定时器16位计数寄存器TPMxCNTH：TPMxCNTL**

TPMxCNTH 为 16 位定时器计数寄存器的高 8 位。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | TPMxCNTH | | | | | | | |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

TPMxCNTL 为 16 位定时器计数寄存器的低 8 位。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | TPMxCNTL | | | | | | | |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

用 8 位总线访问 16 位计数器时必须进行两次操作，在两次操作之间计数器仍然在继续计数。如果直接读取，则高低字节不是同一个时刻的计数值。S08 使用缓冲机制解决这一问题。当读取 16 位计数器的高字节时，其低字节同时被自动锁存到缓冲区中；后面对低字节的读取实际是读取这个缓冲区。而在读取低字节之前，这个缓冲区的内容将保持不变，即使后面又多次读取高字节，该缓冲区仍将保持其原有内容。因此，一个完整的读计数器操作只要读取了高字节就应该读取低字节。高字节也有对应的缓冲区机制。如果先读取计数器的低字节，也一定要读取其高字节。对 16 位计数器进行写入操作时，实际的写入数都被存放在缓冲区中，只有这个 16 位计数器进行写入操作后，该缓冲区的数值才会被实际写入计数器中。

16 位的自由运行计数器在单片机复位结束后从$0000 开始连续计数。当达到最大值$FFFF 时，翻转到$0000，同时将状态寄存器的溢出位 TOF 置 1，然后重新开始计数。在单片机正常工作时，这个计数器会自动计数，但程序也可以停止或者清除其计数。计数器的计数频率是由系统的内部总线频率和计数分频因子来决定的。使用分频因子可以降低计

数器的计数时钟频率，扩大 16 位定时器的计数时间间隔。例如总线频率为 2MHz，分频因子为 32，那么计数器每 16μs 计数一次。如果定时器是自由运行溢出，那么每隔 1.048 s 将产生一次定时器溢出。如果允许定时器溢出中断，将会得到一次定时器的溢出中断服务。

**3. 定时器16位预置计数器寄存器TPMxMODH：TPMxMODL**

TPMxMODH：TPMxMODL 为定时器 16 位预置计数器寄存器的高 8 位。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | TPMxMODH | | | | | | | |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

TPMxMODL 为定时器 16 位预置计数器寄存器的低 8 位。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | TPMxMODL | | | | | | | |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

预置计数器寄存器是一个 16 位寄存器，程序向这个寄存器写入一个确定的数值，则计数器每进行一次计数都会将计数与这个寄存器的值进行比较。如果相同，就产生溢出，计数器翻转到 0 并重新开始计数。也就是说，此时自由运行计数器并没有计数到$FFFF 才产生溢出并重新计数。预置计数器寄存器可以随时进行更改，以得到不同时间间隔的定时器溢出间隔。这里以 4 MHz 晶振且不使用锁频环（2MHz 总线）时钟为例，分频因子为 32。如果预置计数器寄存器设置为$F424（62 500），就可以得到 1s 间隔的定时器溢出，这对于时间计算就非常方便了。

**4. 定时器通道状态和控制寄存器 TPMxCnSC**

TPMxCnSC 包含了定时器通道中断标志、中断请求、时钟源选择和分频因子等功能。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | CHnE | CHnIE | MSnB | MSnA | ELSnB | ELSnA | 0 | 0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

CHnE 为定时器通道溢出标志位。该位用来标志定时器的溢出。当定时器的计数达到$FFFF（自由计数运行状态）或预置计数器的值时，定时器的计数将清零，同时该位置为 1。读取定时器状态和控制寄存器并向该位写入 0，可以清除该标志位。该位虽然是可读/写的，但由于该位只是状态位，向该位写入 1 的操作仅仅完成清除该标志位的作用，没有任何实际意义。另外需要注意，如果清除该标志位前又发生了定时器溢出，那么这一次清零将无法完成。程序必须保证每一次溢出发生后，都要在下一次溢出发生之前清除该标志位。在使用溢出中断的情况下，要求在中断返回之前必须清除溢出标志位，并且该中断的处理时间在最不利的情形下也不能超过一次计数溢出时间。程序在向该位写入 1 进行清除后，可以再次读取

该位，判断是否已经正确清零，这样就能够发现连续的定时器溢出的情况。这种情况往往是因为程序没能及时地处理定时器溢出事件。

CHnIE 为定时器通道溢出中断允许位。该位用来标志是否允许定时器通道溢出中断。该位为 1，表示允许溢出中断，当发生溢出时，系统会执行溢出中断处理程序；该位为 0，表示不允许溢出中断。该位是可读/写的，读取的数值就是其实际值。

MSnB：MSnA 为通道 n 的模式选择位，如表 1-6 所示。

ELSnB：ELSnA 为边沿/电平选择位，如表 1-6 所示。

表 1-6　模式、边沿/电平选择

| CPWMS | MSnB：MSnA | ELSnB：ELSnA | 模　　式 | 配　　置 |
|---|---|---|---|---|
| X | XX | 00 | 引脚不用于定时器，用于通用 I/O 或其他外设控制 | |
| 0 | 00 | 01 | 输入捕捉 | 仅在上升沿捕捉 |
| | | 10 | | 仅在下降沿捕捉 |
| | | 11 | | 在上升沿或下降沿捕捉 |
| | 01 | 01 | 比较输出 | 翻转比较输出 |
| | | 10 | | 清零比较输出 |
| | | 11 | | 置位比较输出 |
| | 1X | 10 | 边沿对齐 PWM | High-true p 脉冲（清零比较输出） |
| | | X1 | | Low-true p 脉冲（置位比较输出） |
| 1 | XX | 10 | 中心对齐 PWM | High-true p 脉冲（清零向上比较输出） |
| | | X1 | | Low-true p 脉冲（置位向上比较输出） |

注：X 表示 0 或 1。

**5. 通道计数器的16位寄存器 TPMxCnVH：TPMxCnVL**

该寄存器包含输入捕捉功能所捕捉的 TPM 计数寄存器值、比较功能或 PWM 功能的比较输出值。该寄存器可通过复位清零。

在输入捕捉模式下，无论从 TPMxCnVH 还是 TPMxCnVL 中读取任何一个字符，都会使两个字节的内容锁入到缓存器中。并且这些内容将会一直被锁定在那里，直到另一个字节被读取。

## 1.2.8　IIC 模块

MC9S08JM60 单片机片内集成了一个 IIC 模块，IIC 模块的两个管脚 SCL、SDA 和 PTC0、PTC1 管脚复用。用户对 IIC 模块的所有操作都是通过该模块的寄存器来完成的。IIC 模块的寄存器主要有 IICA、IICF、IICC1、IICS、IICD、IICC2。下面结合具体寄存器来介绍 IIC 模块的操作。

**1. IIC地址寄存器（IICA）**

IICA 的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | AD7 | AD6 | AD5 | AD4 | AD3 | AD2 | AD1 | 0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

AD[7:1]为从机地址位。AD[7:1]包含 IIC 模块将要使用的从机地址。

**2. IIC分频寄存器（IICF）**

IICF 的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | MULT | | ICR | | | | | |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

MULT 为 IIC 倍频因子位。

00 mul=01

01 mul =02

10 mul=04

11 保留

MULT 定义了倍频因子 mul 的值，mul 和 SCL 的分频因子一起决定 IIC 模块的波特率。

ICR[5:0]为 IIC 模块的时钟频率位。

ICR{5:0}位用于预分频总线时钟，以便进行比特率选择。这些位和 MULT 位一起决定 IIC 模块的波特率、SDA 保持时间、SCL 开始保持时间和 SCL 停止保持时间。

IIC 模块波特率计算公式如下：

IIC 波特率=系统总线频率/（MUL×SCL 分频因子）

SDA 保持时间（SCL 信号的下降沿到 SDA 信号改变之间的延时）计算公式如下：

SDA 保持时间=系统总线周期 × MUL×SDA 保持值

SCL 开始保持时间（当 SCL 处于高电平时，SDA 信号从高到低的跳变时间）的计算公式如下：

SCL 开始保持时间= 系统总线周期× MUL×SCL Start 保持值

SCL 停止保持时间（当 SCL 处于高电平时，SDA 信号从低到高的跳变时间）的计算公式如下：

SCL 停止保持时间= 系统总线周期× MUL×SCL Stop 保持值

**3. IIC控制寄存器1 （IICC1）**

IICC1 的位定义如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | IICEN | IICIE | MST | TX | TXAK | RSTA | 0 | 0 |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

IICEN 为 IIC 模块使能位。

1=允许 IIC 模块。

0=禁止 IIC 模块。

IICIE 为 IIC 中断允许位。

1=允许 IIC 中断请求。

0=禁止 IIC 中断请求。

MST 为主机模式选择位。

1=主机模式。

0=从机模式。

该位从 0 到 1 的改变将产生一个起始信号，此时主机模式被选择。该位从 1 到 0 的改变将产生一个停止信号，此时模块将从主机模式切换为从机模式。

TX 为传输模式选择位。

1=发送。

0=接收。

该位控制主、从传输的方向。在主模式下，该位需按照传输类型来设置。

TXAK 为传输应答使能位。

1=未发送应答响应信号。

0=接收一个字节后向总线发送一个应答响应信号。

RSTA 为重启动位。

在主模式下，该位置为 1 将产生一个重启动条件。

**4. IIC状态寄存器（IICS）**

IICS 的位定义如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | TCF | IAAS | BUSY | ARBL | 0 | SRW | IICIF | RXAK |
| | 写 | X | | X | | | | | |
| 复位 | | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

TCF 为传输完成标志位。

1=传输完成。

0=传输正在进行。

IAAS 为匹配从机标志位。

1=已寻址从机。

0=未寻址。

BUSY 为总线忙标志位。

1=总线忙。

0=总线空闲。

ARBL 为仲裁丢失标志位。

1=仲裁丢失。

0=标准总线操作。

该位表示主/从模式下的总线状态。

SRW 为从机读/写选择位。

1=从机发送，主机读取从机数据。

0=从机接收，主机往从机写数据。

IICIF 为 IIC 中断标志位。

1=中断被挂起。

0=没有中断挂起。

RXAK 为接收应答确认位。

1=没有接收到应答。

0=接收到应答。

**5. IIC数据寄存器（IICD）**

IICD 的位定义如下：

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | DATA | | | | | | | |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

DATA 为传输的数据。

在主发送模式下，写 IICD 寄存器将启动一次数据传输，且最高位先传输；在主机接收模式下，读 IICD 寄存器将启动下一字节的接收。在从模式下，当地址匹配后，可提供相同的作用。

当 IIC 被配置为主接收或从接收模式时，读取 IICD 寄存器将返回接收的最后一个字节。IICD 不反映 IIC 总线上发送的每个字节，软件也不能通过回读的方式确认字节是否已经正确写入 IICD。在主发送模式中，置位 MST 后写入 IICD 的第一个字节数据用于传输地址，应当包含主叫地址（D7～D1）及所需的 R/W 位（D0）。

**6. IIC控制寄存器2（IICC2）**

IICC2 寄存器的各控制位如下所示。X 表示保留。

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | GCAEN | ADEXT | 0 | 0 | 0 | AD10 | AD9 | AD8 |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

GCAEN 为通用呼叫地址使能位。

1=通用呼叫地址使能。

0=禁止通用呼叫地址。

ADEXT 为地址模式选择位。

1=10 位地址模式。

0=7 位地址模式。

该位控制着从机地址使用的位数。

AD[10:9]为从机地址位。

AD[10:8]包含 10 位地址模式中从机地址的高 3 位。当 ADEXT 置为 1 时，AD[10:8]才有效。

## 1.2.9　FLASH 模块

MC9S08JM60 单片机的片内 FLASH 模块有一个集成控制器用来控制编程和擦除操作。该集成控制器有 3 个寄存器，用来产生编程和擦除的时序，也用来提供编程和擦除的电压。对 FLASH 的擦除操作都是以页为基础的，而写入操作则是以行或页为基础的。用户对 FLASH 模块的所有操作都是通过该模块的寄存器来完成的。FLASH 模块的寄存器主要有 FCDIV、FOPT、FCNFG、FPROT、FSTAT、FCMD。下面结合具体寄存器来介绍 FLASH 模块的操作。

**1. FLASH时钟分频寄存器（FCDIV）**

FCDIV 是一个 8 位寄存器，用来控制 FLASH 时钟的频率。Bit 7 是只读位，Bit6～Bit0 可以随时读取，但只能写一次。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | DIVLD | PRDIV8 | DIV5 | DIV4 | DIV3 | DIV2 | DIV1 | DIV0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

DIVLD 为分频寄存器加载状态位。

1=FCDIV 寄存器已经初始化，允许擦除和编程操作。

0=FCDIV 寄存器未初始化，禁止擦除和编程操作。

PRDIV8 为 FLASH 时钟预分频位。

1=FLASH 时钟输入为总线时钟的 8 分频。

0=FLASH 时钟为总线时钟。

DIV[5:0]为 FLASH 时钟的分频因子位。

FLASH 的时钟由 PRDIV8 位和 DIV[5：0]位共同确定。计算公式为：

如果 PRDIV8=0，则 FLASH 时钟 $f_{FCLK}$=总线时钟/（DIV[5:0]+1）

如果 PRDIV8=1，则 FLASH 时钟 $f_{FCLK}$=总线时钟/（8×（DIV[5:0]+1））

**2. FLASH可选寄存器（FOPT）**

系统上电过程中，FOPT 中的内容被复制到非易失 NVOPT 地址中。FOPT 寄存器值可以随时读取，写入无意义。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | KEYEN | FNORED | 0 | 0 | 0 | 0 | SEC01 | SEC00 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

KEYEN 为后门密钥机制允许位。

1=如果用户固件写入的 8 位值匹配非易失的后门密钥，在下一次 MCU 复位前，安全保护暂时失效。

0=禁止后门密钥访问。

FNORED 为向量重映射允许位。

1=允许向量重映射；

0=禁止向量重映射。

SEC0[1:0]为安全状态码位。该两位决定了 MCU 的安全状态，如表 1-7 所示。

表 1-7　MCU 安全状态表

| SEC01:SEC00 | 描　　述 |
|---|---|
| 0:0 | 安全 |
| 0:1 | 安全 |
| 1:0 | 不安全 |
| 1:1 | 安全 |

**3. FLASH配置寄存器（FCNFG）**

FCNFG 的位定义如下所示：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | KEYACC | 0 | 0 | 0 | 0 | 0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

KEYACC 为访问密钥写允许位。

1=NVBACKKEY 的写操作被认为是写比较密钥。

0=地址 0XFFB0～0XFFB7 的写被认为是启动 FLASH 编程或擦除操作。

**4. FLASH保护寄存器（FPROT）**

FPROT 的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | FPS7 | FPS6 | FPS5 | FPS4 | FPS3 | FPS2 | FPS1 | FPDIS |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FPS[7:1]为 FLASH 块保护寄存器位。

FPDIS 为 FLASHA 块保护允许位。

1=禁止 FLASH 块保护。

0=FPS[7:1]指定 FLASH 块保护的区域。

**5. FLASH状态寄存器（FSTAT）**

FSTAT 的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | FCBEF | FCCF | FPVIOL | FACCERR | 0 | FBLANK | 0 | 0 |
| 复位 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

FCBEF 为 FLASH 命令缓冲器空标志位。

1=编程命令可以写入命令缓冲器中。

0=命令缓冲器满。

FCCF 为 FLASHA 命令完成标志位。

1=所有命令已完成。

0=命令正在操作中。

当命令缓冲器未空，且没有处理的命令时，该标志位被自动置为 1。启动一个新命令时，FCCF 标志自动清零。

FPVIOL 为保护冲突标志位。

1=擦除或编程保护区域。

0=未发现保护冲突。

FACCERR 为访问错误标志位。

1=访问错误发生了。

0=没有访问错误。

FBLANK 为 FLASH 空校验标志位。

1=空校验完成，且 FCCF=1，FBLANK=1 表示 FLASH 已经完全擦除了。

0=空校验完成，且 FCCF=1，FBLANK=0 表示 FLASH 未完全擦除。

**6. FLASH命令寄存器（FCMD）**

FCMD 位定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | FCMD7 | FCMD6 | FCMD5 | FCMD4 | FCMD3 | FCMD2 | FCMD1 | FCMD0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FCMD[7:0]为 FLASH 命令位。

FLASH 命令和 FLASH 命令位的关系如表 1-8 所示。

表 1-8　FLASH 命令表

| 命　　令 | FCMD |
|---|---|
| 空检查 | 0x05 |
| 字节编程 | 0x20 |
| 字节编程—突发模式 | 0x25 |
| 页擦除（512 字节/页） | 0x40 |
| 整片擦除 | 0x41 |

## 1.2.10 SCI 模块

在单片机系统中，串口通信是一个非常重要的部分，通过串口通信实现与 PC 或其他嵌入式设备进行通信。MC9S08JM60 单片机内置两个独立的串行通信接口 SCI。SCI 模块的特点如下：

- 支持全双工模式，标准的不归零数据格式。
- 双缓冲的发送器和接收器，发送器和接收器使能独立。
- 中断驱动或查询操作。
- 硬件校验生成和检查。
- 可编程的 8 位或 9 位数据长度。
- 可选的 13 位 break 字符生成或 11 位 break 字符检测。
- 可选的发送时输出极性。

下面具体对 SCI 模块的结构、SCI 模块的寄存器和工作模式进行介绍。

**1. SCI模块的结构**

SCI 硬件模块包括两个部分：发送器部分和接收器部分。发送器和接收器部分的硬件结构如图 1-5 和图 1-6 所示。

其中，SCI 的发送器主要包括发送控制模块、发送缓冲器、发送移位寄存器、波特率生成器和校验生成模块等。发送器在发送时产生一些状态信息，并可以设置发送中断标志。发送控制模块可以选择相应的工作模式，同时设置相应的引脚。SCI 的接收器主要包括接收缓冲器、接收移位寄存器、奇偶检查模块等。接收器在接收的时候产生一些状态信息，并设置相应的中断标志位。用户对 SCI 模块的所有操作都是通过操作该模块的寄存器来完成的，下面对 SCI 模块的寄存器进行具体介绍。

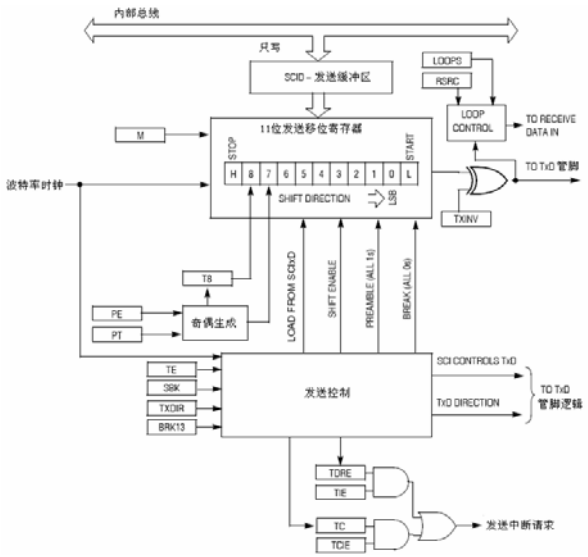



图 1-5　发送器部分的硬件结构图

### 2. SCI模块的寄存器和工作模式

SCI 模块有 8 个 8 位寄存器：SCIxBDH、SCIxBDL、SCIxC1、SCIxC2、SCIxC3、SCIxS1、SCIxS2 和 SCIxD，其中 x 的值可能是 0 或 1，因为 MC9S08JM60 有 2 个 SCI 模块。这 8 个寄存器实现波特率控制、SCI 模式选择、SCI 工作状态报告和发送/接收数据。下面结合具体寄存器来介绍 SCI 模块的功能。

（1）波特率寄存器 SCIxBDH\SCIxBDL

这两个寄存器控制 SCI 系统的波特率。波特率由控制器总线时钟分频而来，发送器和接收器具有相同的波特率和数据格式。

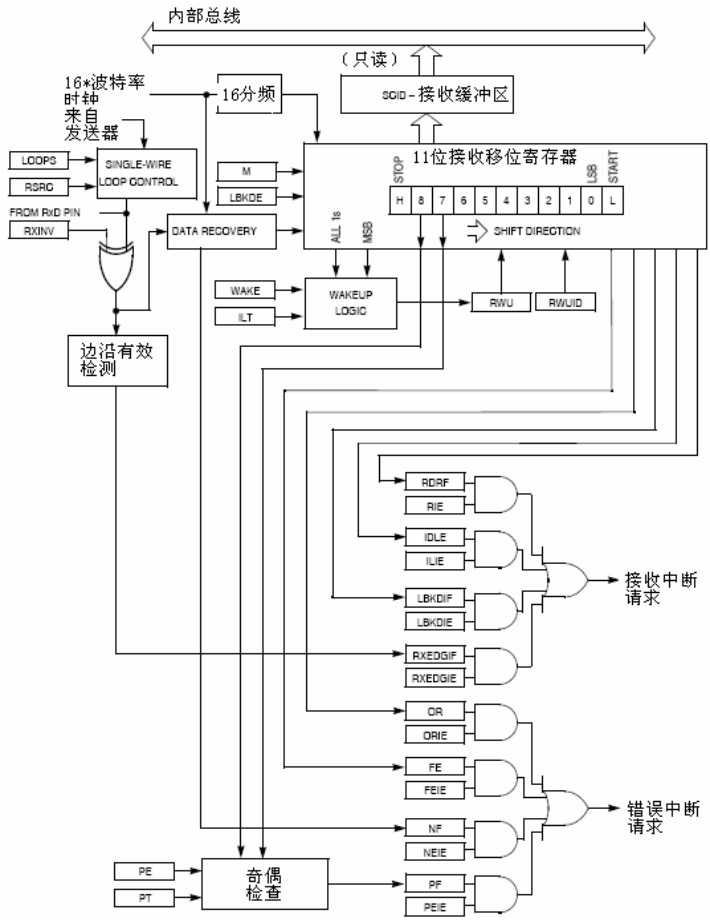



图 1-6　接收器的硬件结构图

其中，SCIxBDH 寄存器描述如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | LBKDIE | RXEDGIE | 0 | SBR12 | SBR11 | SBR10 | SBR9 | SBR8 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

LBKDIE 为 LIN 中断检测中断控制位。

1=硬件中断使能。

0=硬件中断禁止。

RXEDGIE 为 RxD 输入边沿有效中断控制位。

1=硬件中断使能。

0=硬件中断禁止。

SBR[12:8]为波特率预分频因子的高 5 位值。

SCIxBDL 寄存器描述为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | SBR7 | SBR6 | SBR5 | SBR4 | SBR3 | SBR2 | SBR1 | SBR0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

SBR 为波特率预分频因子的低 8 位值。

SCI 波特率计算公式为：系统时钟/(16×SBR[12:0])

（2）SCI 控制寄存器 1（SCIxC1）

SCIxC1 位定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | LOOPS | SCISWAI | RSRC | M | WAKE | ILT | PE | PT |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

LOOPS 为循环模式选择位。

1=循环模式。

0=正常操作。

在循环模式时，RxD 引脚与 SCI 系统断开，发送器的输出直接接入接收器的输入。正常操作时，SCI 的发送引脚和接收引脚独立。

SCISWAI 为等待模式下 SCI 停止控制位。

1=等待模式下，SCI 时钟停止。

0=等待模式下，SCI 时钟继续运行。

RSRC 为接收源选择位。

1=SCI 在循环模式下，TxD 引脚接发送器的输出和接收器的输入。

0=SCI 在循环模式下，SCI 不使用 RxD 引脚。

当 LOOPS 为 0 时，该位无任何意义。

M 为字符长度选择位。

1=接收器和发送器使用 9 位字符长度。

0=接收器和发送器使用 8 位字符长度。

WAKE 为唤醒条件选择位。

1=地址唤醒。

0=空闲线唤醒。

ILT 为空闲线方式选择位。

1=从停止位开始计数。

0=从起始位开始计数。

PE 为校验使能位。

1=校验使能。

0=无硬件校验或检查。

PT 为校验类型选择位。

1=奇校验。

0=偶校验。

（3）SCI 控制寄存器 2（SCIxC2）

SCIxC2 寄存器是 SCI 子系统的主控制器寄存器。这个寄存器可以允许和禁止发送器或接收器，允许和禁止系统中断、唤醒功能和发送中止码功能。TIE、TCIE、RIE、ILIE 位是局部中断控制，决定 SCI 系统是查询状态标志还是发出硬件中断请求。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | TIE | TCIE | RIE | ILIE | TE | RE | RWU | SBK |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

TIE 为发送中断允许位。

1=发送中断允许。

0=禁止发送中断。

TCIE 为发送完成中断允许位。

1=允许发送完成中断。

0=禁止发送完成中断。

RIE 为接收终端允许位。

1=允许接收中断。

0=禁止接收中断。

ILIE 为空闲中断允许位。

1=允许空闲中断。

0=禁止空闲中断。

TE 为发送允许位。

1=允许发送器。

0=禁止发送器。

RE 为接收允许位。

1=允许接收器。

0=禁止接收器。

RWU 为接收唤醒位。

1=接收器处于等待状态，并关闭接收中断。当 SCIxC1 的 WAKE 位决定的唤醒条件发生时，接收器退出等待状态，并清零 RWU 位。

0=接收器正常工作。

SBK 为终止位。

1=发送终止码。

0=发送器正常工作。

（4）SCI 控制寄存器 3（SCIxC3）

SCIxC3 位定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | R8 | T8 | TXDIR | TXINV | ORIE | NEIE | FEIE | PEIE |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

R8 为接收位 8。

当选择 9 位字符时，该位为接收的最后一位。

T8 为发送位 8。

当选择 9 位字符时，该位为发送的最后一位。

TXDIR 为单线模式下 TxD 管脚方向位。

1=单线模式下，TxD 管脚为输出。

0=单线模式下，TxD 管脚为输入。

TXINV 为发送数据反向位。

1=发送数据反向。

0=正常数据发送。

ORIE 为接收器溢出中断允许位。

1=允许接收器溢出中断。

0=禁止接收器溢出中断。

NEIE 为接收器噪声错误中断允许位。

1=允许接收器噪声错误中断。

0=禁止接收器噪声错误中断。

FEIE 为接收器帧错误中断允许位。

1=允许接收器帧错误中断。

0=禁止接收器帧错误中断。

PEIE 为接收器奇偶错误中断允许位。

1=允许接收器奇偶错误中断。

0=禁止接收器奇偶错误中断。

（5）SCI 通信状态寄存器 1（SCIxS1）

SCIxS1 位定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | TDRE | TC | RDRF | IDLE | OR | NF | FE | PF |
| 复位 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

TDRE 为发送数据寄存器空标志。

1=发送数据寄存器空。

0=发送数据寄存器满。

TC 为发送完成标志。

1=发送器空闲。

0=发送器忙。

RDRF 为接收数据寄存器满标志。

1=接收数据寄存器满。

0=接收数据寄存器空。

读出 SCIxS1，再读出 SCDR 时，清空该标志位。

IDLE 为接收器空闲标志。

1=空闲线被检测到。

0=没有检测到空闲线。

读出 SCIxS1，再读出 SCDR 时，清空该标志位。

OR 为接收器溢出标志。

1=接收器溢出。

0=接收器未溢出。

读出 SCIxS1，再读出 SCDR 时，清空该标志位。

NF 为接收器噪声标志位。

1=接收器噪声被检测到。

0=接收器噪声未检测到。

FE 为接收器帧错误标志位。

1=接收器帧错误。

0=接收器帧无错。

读出 SCIxS1，再读出 SCDR 时，清空该标志位。

PF 为接收器奇偶错标志位。

1=接收器奇偶错。

0=接收器奇偶正确。

读出 SCIxS1，再读出 SCDR 时，清空该标志位。

（6）SCI 通信状态寄存器 2（SCIxS2）

SCIxS2 位定为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | LBKDIF | RXEDGIF | 0 | RXINV | RWUID | BRK13 | LBKDE | RAF |
| 复位 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

LBKDIF 为 LIN 终止码中断标志。

1=LIN 终止码中断。

0=LIN 终止码未发生中断。

写 1 清空该标志位。

RXEDGIF 为 RxD 管脚边沿有效中断标志位。

1=边沿有效中断。

0=边沿有效未发生中断。

写 1 清空该标志位。

RXINV 为接收数据反向。

1=接收数据反向。

0=接收数据未反向。

RWUID 为空闲唤醒接收检测位。

1=空闲字符唤醒接收。

0=未发现空闲字符唤醒接收。

BRK13 为终止码生成长度控制位。

1=13 位时间长度的终止码正在发送。

0=10 位时间长度的终止码正在发送。

LBKDE 为 LIN 终止码检测控制位。

1=允许 11 位时间长度的终止码检测。

0=允许 10 位时间长度的终止码检测。

RAF 为接收器帧错误标志位。

1=接收器帧错误。

0=接收器帧无错。

读出 SCIxS1，再读出 SCDR 时，清空该标志位。

（7）SCI 通信数据寄存器 2（SCIxD）

SCIxD 位定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | R7 | R6 | R5 | R4 | R3 | R2 | R1 | R0 |
| 定义 | T7 | T6 | T5 | T4 | T3 | T2 | T1 | T0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

串行通信数据寄存器（SCIxD）具有双重功能：读时是接收寄存器；写时是发送寄存器。

HCS08 系列单片机的 SCI 模块采用了发送器和接收器双缓冲技术。SCI 的硬件缓冲区有两个字节，串行通信数据寄存器可以存储一个字节，移位寄存器可以存储一个字节。即使 CPU 因处理单个字符而延迟回应，也可以很容易地处理一串连续字符，因此用户在编制串行通信软件时有很大的回旋余地。

### 1.2.11　SPI 模块

MC9S08JM60 单片机内置两个同步串行外设接口，即 SPI。SPI 系统用于同标准外设芯片通信。这类芯片主要包括其他单片机、模数转换器、移位寄存器、传感器、存储器等。

一个典型的 SPI 系统包括一个主微控制器和一个或多个从属外设。很多制造商都生产同步串行外设，为了兼容大多数外设，SPI 系统可以选择移位率、主从模式及时钟的极性、相位等。

MC9S08JM60 的 SPI 模块提供了 3 类寄存器：控制、状态和数据寄存器。其中控制类寄存器有：SPIxC1、SPIxC2、SPIxBR、SPIxMH 和 SPIxML；状态类寄存器有：SPIxS；数据类寄存器有：SPIxDH 和 SPIxDL。下面结合具体的寄存器来介绍 SPI 模块。

**1. SPI模块的控制类寄存器**

控制类寄存器较多，主要包括 SPIxC1、SPIxC2、SPIxBR、SPIxMH 和 SPIxML5 个寄存器。这 5 个寄存器主要控制 SPI 模块的工作。下面分别对这 5 个寄存器进行介绍。

（1）SPI 控制寄存器 1（SPIxC1）

SPIxC1 的位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | SPIE | SPE | SPTIE | MSTR | CPOL | CPHA | SSOE | LSBFE |
| 复位 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

SPIE 为 SPI 中断使能位。

1=允许 SPI 中断。

0=禁止 SPI 中断。

SPE 为 SPI 模块允许位。

1=允许 SPI 模块。

0=禁止 SPI 模块。

SPTIE 为 SPI 发送中断允许位。

1＝允许 SPI 在 SPTE 位被置为 1 时产生中断请求。

0＝禁止 SPI 在 SPTE 位被置为 1 时产生中断请求。

MSTR 为 SPI 主机位。

1=选择 SPI 主机模式。

0=选择 SPI 从机模式。

CPOL 为时钟奇偶位。

1=选择低电平有效时钟，SPI 时钟空闲状态为高电平。

0=选择高电平有效时钟，SPI 时钟空闲状态为低电平。

CPHA 为时钟相位位。

1=从机以 SPSCK 的第一次沿跳变为移位开始信号。

0=从机以/SS 的下降沿作为移位开始信号。

SSOE 为从模式输出使能。

该位和其他控制位一起决定/SS 管脚的功能。

LSBFE 为移位方向位。

1=SPI 串行数据传输以最低位开始。

0=SPI 串行数据传输以最高位开始。

（2）SPI 控制寄存器 2（SPIxC2）

SPIxC2 位定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | SPMIE | SPIMODE | 0 | MODFEN | BIDIROE | 0 | SPISWAI | SPC0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

SPMIE 为 SPI 匹配中断允许位。

1=允许 SPI 匹配中断。

0=禁止 SPI 匹配中断。

SPIMODE 为 SPI 模式选择位。

1=允许 SPI 模块。

0=禁止 SPI 模块。

MODFEN 为模式设置错误检测允许位。

1＝允许 MODF 标志置为 1。

0＝禁止 MODF 标志置为 1。

BIDIROE 为双向模式输出允许位。

1=SPI I/O 管脚为输出。

0=输出驱动器关闭，SPI 数据 I/O 管脚为输入。

SPISWAI 为等待模式下 SPI 的停止位。

1=MCU 进入等待模式，SPI 时钟关闭。

0=MCU 进入等待模式，SPI 时钟继续工作。

SPC0 为 SPI 管脚控制位。

1=配置 SPI 为单线双向工作。

0=SPI 使用独立的数据输入/输出管脚。

（3）SPI 波特率寄存器（SPIxBR）

SPIxBR 位定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | SPPR2 | SPPR1 | SPPR0 | 0 | SPR2 | SPR1 | SPR0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

SPPR[2:0]为 SPI 移位频率预分频选择位。这 3 位定义波特率预分频系数，如表 1-9 所示。预分频的输入是总线时钟（BUSCLK）；预分频器的输出驱动 SPI 波特率系数的输入。

表 1-9　SPI 波特率预分频系数

| SPPR2:SPPR1:SPPR0 | 预分频因子 |
|---|---|
| 0:0:0 | 1 |
| 0:0:1 | 2 |
| 0:1:0 | 3 |
| 0:1:1 | 4 |
| 1:0:0 | 5 |
| 1:0:1 | 6 |
| 1:1:0 | 7 |
| 1:1:1 | 8 |

SPR[2:0]为 SPI 移位频率分频选择位。这 3 位定义 SPI 波特率分频系数，如表 1-10 所示。该被除数的输入来自 SPI 波特率预分频器；输出是主模式的 SPI 波特率时钟。

表 1-10　SPI 波特率分频系数

| SPR2:SPR1:SPR0 | 分频因子 |
|---|---|
| 0:0:0 | 2 |
| 0:0:1 | 4 |
| 0:1:0 | 8 |
| 0:1:1 | 16 |
| 1:0:0 | 32 |
| 1:0:1 | 64 |
| 1:1:0 | 128 |
| 1:1:1 | 256 |

（4）SPI 匹配寄存器（SPIxMH）

SPIxMH 的位定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | Bit15 | 14 | 13 | 12 | 11 | 10 | 9 | Bit8 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

（5）SPI 匹配寄存器（SPIxML）

SPIxML 的位定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | Bit7 | 6 | 5 | 4 | 3 | 2 | 1 | Bit0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

SPI 匹配寄存器 SPIxMH:SPIxML 存储硬件的比较值。当 SPI 接收数据缓冲器的值和匹配寄存器的值相同时，匹配标志 SPMF 将置为 1。SPI 选择 8 位模式时，仅 SPIxML 匹配寄存器可用。

**2. SPI模块的状态类寄存器**

状态类寄存器只有一个寄存器——SPIxS。SPIxS 有 4 个状态标志，用来指示 SPI 模块设置和数据传输过程中的状态。这些状态被 SPI 事件自动置位，由软件进行清零。

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | SPRF | SPMF | SPTEF | MODF | 0 | 0 | 0 | 0 |
| 复位 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |

SPRF 为 SPI 接收结束位。

1=接收数据缓冲器满。

0=接收数据缓冲器未满。

当单片机和外部设备之间的数据传输结束后，数据从移位寄存器进入接收数据寄存器中，单片机将 SPRF 置 1 以通知用户接收结束。SPRF 被置 1 后，用户在程序里读一次 SPIxS，将清零该位。

SPMF 为 SPI 匹配标志位。

1=SPI 接收数据缓冲器的值和 SPI 匹配寄存器中的值相等。

0= SPI 接收数据缓冲器的值和 SPI 匹配寄存器中的值不相等。

SPTEF 为 SPI 发送缓冲器空标志位。

1=SPI 发送数据缓冲器空。

0= SPI 发送数据缓冲器未空。

在单片机和外部设备之间的数据传输过程中，数据从发送数据寄存器进入移位寄存器中，单片机将 SPTER 置 1 以通知用户发送结束。读 SPIxS 寄存器，清零该位。

MODF 为模式设置错误标志位。

1=/SS 引脚逻辑电平不正确。

0=/SS 引脚逻辑电平正确。

当 MODF 为 1，且/SS 引脚信号与 SPI 当前工作方式（主机或从机）不符时，具体地说，就是当从机的/SS 为高电平，或是主机的/SS 为低电平时，MODF

会置 1。MODF 被置 1 后，用户在程序里读一次 SPIxS 寄存器，再对 SPIxC1 进行一次写操作，会清零 MODF 位。

**3. SPI模块的数据类寄存器**

数据类寄存器包含两个寄存器：SPIxDH 和 SPIxDL。

（1）SPI 数据寄存器 SPIxDH

SPIxDH 的位定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | Bit 15 | 14 | 13 | 12 | 11 | 10 | 9 | Bit 8 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

（2）SPI 数据寄存器 SPIxDL

SPIxDL 的位定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | Bit 7 | 6 | 5 | 4 | 3 | 2 | 1 | Bit 0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

SPI 数据寄存器（SPIxDH:SPIxDL）具有双重功能：读时是接收寄存器；写时是发送寄存器。

在实际的工作过程中，数据一旦写入发送数据寄存器，就会立即送到移位寄存器并将 SPTEF 置 1。下一次数据发送之前必须先清零 SPTER 标志位，否则发送数据将被忽略。

在数据接收过程中，SPRF 标志置 1，则用户可以读取数据。前一个数据未读取，继续接收第二个数据将会产生数据溢出错误，同时第二个数据将会丢失。

## 1.2.12　ADC 模块

A/D 转换模块（Analog To Digital Convert Module，模/数转换模块）的功能是将电压信号转换为对应的数字信号。实际应用中，这个电压信号可能由温度、湿度、压力等实际物理量经过传感器和相应的变换电路转化而来。MC9S08JM60 单片机集成了两个 12 通道 12 位模数转换器 ADC12V1，A/D 转换模块的输入引脚和 I/O 端口引脚复用。ADC12V1 模块主要有以下特点：

- AD 转换结果可选 12 位、10 位或 8 位无符号格式。
- 单次或连续转换模式。
- 可配置的采样时间和转换速度/功耗。
- AD 时钟可从 4 种时钟源中任选其一。
- 在低噪声应用中，可工作在等待模式或 stop3 模式下。
- 可选的异步硬件转换触发器。

- 自动比较中断功能。
- 温度传感器功能。

用户对 ADC12V1 模块的所有操作都是通过该模块的寄存器完成的。ADC12V1 模块的寄存器大致可分为以下 4 类：状态和控制类、数值类、配置类，以及管脚控制类。其中状态和控制类的寄存器有：ADCSC1 和 ADCSC2；数值类寄存器有：转换结果寄存器 ADCRH 、ADCRL 和转换比较值寄存器 ADCCVH、ADCCVL；配置类寄存器有：ADCCFG；管脚控制类寄存器有：APCTL1、APCTL2、APCTL3。下面结合具体的寄存器来介绍 ADC12V1 模块。

**1. ADC12V1模块的状态和控制类寄存器**

状态和控制类的寄存器主要包括 ADCSC1 和 ADCSC2 两个寄存器。这两个寄存器主要控制 ADC12V1 模块的工作。下面分别对这两个寄存器进行介绍。

（1）A/D 状态和控制寄存器 ADCSC1

ADCSC1 位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | COCO | AIEN | ADCO | ADCH | | | | |
| 复位 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

COCO 为 A/D 转换标志完成位。

这个只读标志位在每次转换完成后自动置 1。当写入 ADCSC1 或者读取数据结果寄存器（ADCRL）时，将清除该标志位。

AIEN 为 A/D 中断允许控制位。

1=允许 A/D 中断，系统将在转换完成时产生中断（由 COCO 出发）。

0=禁止 A/D 中断。

ADCO 为 A/D 连续转换控制位。

1＝连续转换模式，A/D 转换器会不断地进行连续转换，并在每次转换结束后将新的转换结果写入 A/D 数据寄存器中；

0＝单次转换模式，在写入 ATDISC 后只触发一次 A/D 转换。

ADCH 为 A/D 输入信道选择控制位。其对应关系如表 1-11 所示。

表 1-11　ATDCH 与输入信道选择的对应关系

| ATDCH | 输入信道选择 |
|---|---|
| 00000–01111 | AD0-15 |
| 10000–11011 | AD16-27 |
| 11100 | 保留 |
| 11101 | $V_{REFH}$ |
| 11110 | $V_{REFL}$ |
| 11111 | 模块禁止使用 |

（2）A/D 状态和控制寄存器 ADCSC2

A/D 状态和控制寄存器 ADCSC2 控制 A/D 转换的比较功能、转换触发和 A/D 模块转换的启动等。

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | ADACT | ADTRG | ACFE | ACFGT | 0 | 0 | | |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

ADACT 为 A/D 转换有效状态位。

1=A/D 转换正在进行。

0=A/D 转换未进行。

该标志位在 A/D 转换模块初始化后置 1，在 A/D 转换结束或终止时清零。

ADTRG 为 A/D 转换触发选择位。

1=硬件触发 A/D 转换。

0=软件触发 A/D 转换。

该位启动 A/D 转换的触发类型。选择软件触发方式时，对 ADCSC1 的写操作立即启动 A/D 转换；选择硬件触发方式时，ADHWT 管脚上的高电平输入立即启动 A/D 转换。

ACFE 为 A/D 模块比较功能控制位。

1＝使能比较功能。

0＝禁止比较功能。

ACFGT 为 A/D 模块比较功能超出功能控制位。

1＝信道输入值的转换结果大于或等于比较值时，触发比较功能。

0＝信道输入值的转换结果小于比较值时，触发比较功能。

**2. ADC12V1模块的数值类寄存器**

ADC12V1 模块的数值类寄存器主要存储 A/D 转换的结果和比较值。主要包含 ADCRH 、ADCRL 和转换比较值寄存器 ADCCVH、ADCCVL。下面对这 4 个寄存器分别进行介绍。

ADCRH 寄存器存储 A/D 转换结果的高位值。当 A/D 转换精度选择为 12 位时，ADCRH 存储 12 位转换结果的高 4 位；当 A/D 转换精度选择为 10 位时，ADCRH 存储 10 位转换结果的高 2 位，ADR11～ADR10 位被清零；当 A/D 转换精度选择为 8 位时，ADCRH 的 ADR11～ADR8 位被清零。

（1）ADCRH 寄存器

定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | 0 | 0 | ADR11 | ADR10 | ADR9 | ADR8 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

（2）ADCRL 寄存器

ADCRL 寄存器存储 A/D 转换结果的低 8 位值。定义为：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | ADR7 | ADR6 | ADR5 | ADR4 | ADR3 | ADR2 | ADR1 | ADR0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

（3）ADCCVH 寄存器

ADCCVH 寄存器存储 A/D 转换比较值的高位值。当 A/D 转换精度选择为 12 位时，使能比较功能后，ADCCVH 存储 12 位比较值的高 4 位；当 A/D 转换精度选择为 10 位时，使能比较功能后，ADCCVH 存储 10 位比较值的高 2 位；当 A/D 转换精度选择为 8 位时，ADCCVH 寄存器的值不使用。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | 0 | 0 | ADCV11 | ADCV10 | ADCV9 | ADCV8 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

（4）ADCCVL 寄存器

ADCCVL 寄存器存储 A/D 转换比较值的低 8 位值。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | ADR7 | ADR6 | ADR5 | ADR4 | ADR3 | ADR2 | ADR1 | ADR0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**3. ADC12V1模块的配置类寄存器**

配置类寄存器只包括 ADCCFG 一个寄存器。这个寄存器主要控制 ADC12V1 模块的时钟源、分频因子和工作方式等。具体描述如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | ADLPC | ADIV | | ADLSMP | MODE | | ADICLK | |
| 复位 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

ADLPC 为低功耗配置位。

1=高速配置。

0=低功耗配置。

该位控制逐次逼近 A/D 转换的速度和功耗。当不需要高采样率时，可使用低功耗配置来降低系统功耗。

ADIV 为 A/D 时钟分频因子控制位。

该位设置 A/D 转换的时钟分频因子。时钟分频因子值及其对应的总线频率如表 1-12 所示。

表 1-12　分频率选择

| ADIV | 分频因子 | A/D 时钟频率 |
|---|---|---|
| 00 | 1 | A/D 输入时钟 |
| 01 | 2 | A/D 输入时钟/2 |
| 10 | 4 | A/D 输入时钟/4 |
| 11 | 8 | A/D 输入时钟/8 |

ADLSMP 为 A/D 采样时间控制位。

1=长采样时间。

0=短采样时间。

该位控制逐次逼近 A/D 转换的采样时间，利用该位可实现高阻抗输入值的精确采样和低阻抗输入的快速采样。长采样时间也有利于降低系统功耗。

MODE 为 A/D 转换模式控制位。

该位控制 A/D 转换精度，可配置选择 12 位、10 位和 8 位转换精度。模式控制位及其对应的转换精度如表 1-13 所示。

表 1-13　转换模式选择表

| MODE | 转换精度（位） |
|---|---|
| 00 | 8 |
| 01 | 12 |
| 10 | 10 |
| 11 | 保留 |

ADICLK 为 A/D 转换输入时钟控制位。

该位控制 A/D 转换输入时钟源。ADICLK 控制位及其对应的时钟源如表 1-14 所示。

表 1-14　输入时钟源选择

| ADICLK | A/D 转换输入时钟源 |
|---|---|
| 00 | 系统时钟 |
| 01 | 系统时钟/2 |
| 10 | 备选时钟 ALTCLK |
| 11 | 异步时钟 ADACK |

**4. ADC12V1模块的管脚控制类寄存器**

管脚控制类寄存器包括 APCTL1、APCTL2 和 APCTL3 共 3 个寄存器。这 3 个寄存器主要控制 ADC12V1 模块的模拟输入通道和通用 I/O 复用。下面分别对这 3 个寄存器进行介绍。

（1）管脚控制寄存器 APCTL1

管脚控制寄存器 APCTL1 控制 ADC 模块的 0～7 号模拟输入通道和通用 I/O 的复用。各控制位如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | ADPC7 | ADPC6 | ADPC5 | ADPC4 | ADPC3 | ADPC2 | ADPC1 | ADPC0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

ADPC7～ADPC0 为 ADC 引脚允许控制位。

1＝允许该引脚为 A/D 转换模块使用。

0＝不允许该引脚为 A/D 转换模块使用。

（2）管脚控制寄存器 APCTL2

管脚控制寄存器 APCTL2 控制 ADC 模块的 8～15 号模拟输入通道和通用 I/O 的复用。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | ADPC15 | ADPC14 | ADPC13 | ADPC12 | ADPC11 | ADPC10 | ADPC9 | ADPC8 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

ADPC15～ADPC8 为 ADC 引脚允许控制位。

1＝允许该引脚为 A/D 转换模块使用。

0＝不允许该引脚为 A/D 转换模块使用。

（3）管脚控制寄存器 APCTL3

管脚控制寄存器 APCTL3 控制 ADC 模块的 16～23 号模拟输入通道和通用 I/O 的复用。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | ADPC23 | ADPC22 | ADPC21 | ADPC20 | ADPC19 | ADPC18 | ADPC17 | ADPC16 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

ADPC23～ADPC16 为 ADC 引脚允许控制位。

1＝允许该引脚为 A/D 转换模块使用。

0＝不允许该引脚为 A/D 转换模块使用。

### 1.2.13 USB 模块

MC9S08JM60 单片机内置了通用串行总线设备控制器（S08USBV1）模块，即 USB。它基于通用串行总线规格版本 2.0。USB 模块提供了单片全速（12Mbps）USB 设备应用的解决方案，而且集成了必需的带有串行接口引擎（SIE）的收发器、3.3V 稳压器、端点 RAM 和其他控制逻辑。下面具体对 USB 模块的结构、USB 寄存器和工作模式进行介绍。

**1. USB模块的结构**

由图 1-7 可以看出，USB 模块主要由串行接口引擎（SIE）、USB RAM 及其管理模块、

SkyBlue Gasket 接口、USB 收发器、片上调压器等部分组成。用户对 USB 模块的所有操作都是通过操作该模块的寄存器完成的。下面将介绍 USB 模块的寄存器。

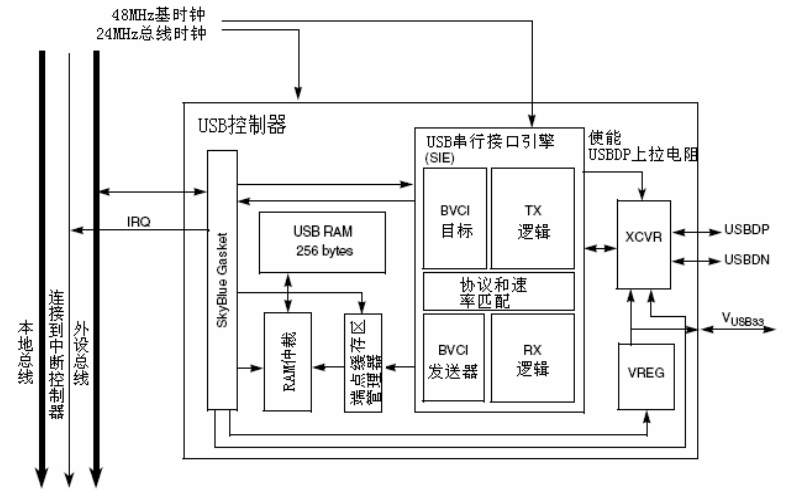



图 1-7　USB 模块的硬件结构图

**2. USB模块的寄存器**

USB 模块的寄存器较多，主要分为控制类寄存器和状态类寄存器。下面结合具体的寄存器来介绍 USB 模块的功能。

（1）USB 控制寄存器 0（USBCTL0）

位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | USBRESET | USBPU | USBRESMEN | LPRESF | 0 | USBVREN | 0 | USBPHYEN |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

USBRESET 为 USB 复位状态。

1=USB 模块返回它的复位状态。

0=USB 模块正常运行。

该位产生一个 USB 模块硬复位，USBPHYEN 和 USBVREGEN 位也会被清除。当置为 1 时，该位在复位时自动清除。

USBPU 为上拉源控制位。

1=内部 USBDP 上拉电阻允许。

0=内部 USBDP 上拉电阻被禁用，此时的应用可使用一个外部的上拉电阻。

USBRESMEN 为 USB 低电压唤醒事件允许控制位。

1=USB 从挂起模式异步唤醒允许。

0=USB 从挂起模式异步唤醒禁止。

该位置为 1，当 USB 模块检测到 LPRESF 位置为 1，那么 USB 模块会向 MCU 提出一个异步唤醒中断。

LPRESF 为低电压恢复标志位。

1=当设备处在 stop3 而且 USB 处在挂起状态，并当 USBRESMEN=1 时，USB 总线上检测到 K-状态。

0=当设备处在 stop3 而且 USB 处在挂起状态，USB 总线上没有检测到 K-状态。

在 USB 挂起模式下，如果 USBRESMEN=1 而且 K-状态在 USB 总线上被检测到，该位置为 1，表明设备在低电压 stop3 模式下发出了恢复信号。该标志位将会触发一个异步中断，这个中断将会把设备从 stop3 模式下唤醒。为了清除 LPRESF 标志位，固件必须先清除 USBRESMEN 位。

USBVREN 为 USB 电压调节器使能。

1=片上 3.3V USB 电压调节器允许，在主模式或备用模式下。

0=片上 3.3V USB 电压调节器被禁用。

USBPHYEN 为 USB 收发器允许。

1=片上 XCVR 被使能。

0=片上 XCVR 被禁止。

当 USB 收发器（XCVR）被禁用时，USBDP 和 USBDN 是 hi-Z。推荐设置 CTL 寄存器的 USBEN 位为 1，前置 XCVR 位为 1。固件必须确保 XCVR 在进入 USB 挂起模式时保持使能状态。

（2）外设 ID 寄存器（PERID）

PERID 寄存器的读回值为 0x04，这个是为 USB 模块外设而定义的。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | ID5 | ID4 | ID3 | ID2 | ID1 | ID0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

ID[5:0]为外设配置编号。这个编号被设置为 0x04，表示外设处在全速 USB 模块状态下。

（3）外设 ID 反码寄存器（IDCOMP）

读 IDCOMP 寄存器将返回外设 ID 寄存器的反码值。对于 USB 模块外设此值应该是 0Xfb。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | NID5 | NID4 | NID3 | NID2 | NID1 | NID0 |
| 复位 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

NID[5:0]为 ID 编码的反码，即 ID[5:0]的反码。

（4）外设版本寄存器（REV）

读 REV 寄存器将返回外设 USB 的外设版本号值。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | REV7 | REV6 | REV5 | REV4 | REV3 | REV2 | REV1 | REV0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

REV[7:0]为 USB 模块的版本号。

（5）中断状态寄存器（INTSTAT）

INTSTAT 寄存器包含了 USB 模块中断源的每一位。这些位由它们各自的中断允许位控制。该寄存器的所有位是从单个的中断源“逻辑或”到一起的。一旦一个中断位被置位，它仅能够通过向相应的中断位写 1 来清除。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | STALLF | 0 | RESUMEF | SLEEPF | TOKDNEF | SOFTOKF | ERRORF | USBRSTF |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

STALLF 为 STALLF 标志。

1=一个 STALL 握手被发出。

0=一个 STALL 握手未发出。

STALL 中断被用在设备模式下。在设备模式下，当一次 STALL 握手被串行接口引擎（SIE）发送时，STALL 标志被置为 1。

RESUMEF 为恢复标志位。

1=检测到恢复信号。

0=没有检测到恢复信号。

在 USB 模块的时钟伴随恢复信号被重新启动后 2.5us 时，该位被置 1。表明 USB 总线上有远程唤醒信号。

SLEEPF 为睡眠标志位。

1=USB 检测到一个持续 3ms 稳定的空闲状态。

0=USB 总线上没有检测到 3ms 空闲状态。

TOKDNEF 为令牌完成标志位。

1=当前处理的令牌已经完成。

0=无处理完成的令牌操作。

当前传送完成时此位被置 1。固件必须立即读寄存器 STAT，以确定端点和 BD 信息。清除此位（通过向此位写 1）将引起 STAT 寄存器被清除或者 STAT FIFO 保持寄存被存入 STAT 寄存器。

SOFTOKF 为 SOF 令牌标志位。

1=USB 模块接收到 SOF 令牌。

0=USB 模块未接收到 SOF 令牌。

ERRORF 为错误标志位。

1=检测到 ERRSTAT 寄存器中的错误条件。

0=没有检测到 ERRSTAT 寄存器中的错误条件。

USBRSTF 为 USB 复位标志位。

1=检测到 USB 复位信号。

0=没有检测到 USB 复位。

当 USB 模块解析到一个合法的 USB 复位时该位将被置 1。当确定后，该位会通知 MCU 自动向地址寄存器写 0x00 并且使能端点 0。一旦检测到持续 2.5us 的复位信号，USBRSTF 将置 1。

（6）中断允许寄存器（INTENB）

寄存器 INTENB 包含了 USB 模块的每一个中断源允许位。设置这些位中的任何一位将使能 INTENB 寄存器中的相应中断源。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | STALL | 0 | RESUME | SLEEP | TOKDNE | SOFTOK | ERROR | USBRST |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

STALL 为 STALL 中断允许位。

1=允许 STALL 中断。

0=禁止 STALL 中断。

RESUME 为恢复中断允许位。

1=允许恢复中断。

0=禁止恢复中断。

SLEEP 为睡眠中断允许位。

1=允许睡眠中断。

0=禁止睡眠中断。

TOKDNE 为 TOKDNE 中断允许位。

1=允许 TOKDNE 中断。

0=禁止 TOKDNE 中断。

SOFTOK 为 SOF 令牌中断允许位。

1=允许 SOF 令牌中断。

0=禁止 SOF 令牌中断。

ERROR 为错误中断允许位。

1=允许错误中断。

0=禁止错误中断。

USBRST 为 USB 复位中断允许位。

1=允许 USB 复位中断。

0=禁止 USB 复位中断。

（7）错误中断状态寄存器（ERRSTAT）

寄存器 ERRSTAT 的每一位包含了 USB 模块的相应错误源。这些位中的每一位都对应相应的错误允许位。这些结果被“与”在一起，然后发送到 INTSTAT 寄存器的错误位。一旦一个中断位被置 1，向相应的标志位写 1 可以清除它。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | BTSERRF | 0 | BUFERRF | BTOERRF | DFN8F | CRC16F | CRC5F | PIDERRF |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

BTSERRF 为位填充错误标志位。

1=位填充标志置位。

0=没有检测到位填充错误。

BUFERR 为缓冲区错误标志位。

1=检测到缓冲错误。

0=没有检测到缓冲错误。

如果 USB 模块已经请求了读存储器中新的 DB，但是在 USB 模块需要接收或传输数据前没有得到总线控制权，该位将置 1。如果处理一次 TX（在端点内）传输，将会引起一次传输数据下溢条件。或者如果处理一次 RX（在端点外）传输，将会引起一次接收数据下溢条件。如果发送给或者从机接收到的数据包比缓冲区大，被分配到 BD，该位也将置 1。

BTOERRF 为总线转向超时错误位。

1=发生了总线转向超时错误。

0=没有总线转向超时错误发生。

DFN8F 为数据字段错误标志位。

1=数据字段不是整数字节。

0=数据字段是整数字节。

数据字段接收到的不是 8 位间隔。USB 规则规定了，数据字段必须是整数个字节。如果数据字段不是整数个字节，该位将置 1。

CRC16F 为 CRC16 错误标志位。

1=检测到 CRC16 错误。

0=没有检测到 CRC16 错误。

CRC5F 为 CRC5 错误标志位。

1=检测到 CRC5 错误，令牌包将被丢弃。

0=没有检测到 CRC5 错误。

PIDERRF 为 PID 错误标志位。

1=检测到 PID 校验错误。

0=没有检测到 PID 校验错误。

（8）错误中断允许寄存器（ERRENB）

位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | BTSERR | 0 | BUFERR | BTOERR | DFN8 | CRC16 | CRC5 | PIDERR |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

BTSERR 为 BTSERR 中断允许位。

1=允许 BTSERR 中断。

0=禁止 BTSERR 中断。

BUFERR 为 BUFERR 中断允许位。

1=允许 BUFFRR 中断。

0=禁止 BUFERR 中断。

BTOERR 为 BTOERR 中断允许位。

1=允许 BTOERR 中断。

0=禁止 BTOERR 中断。

DFN8 为 DFN8 中断允许位。

1=允许 DFN8 中断。

0=禁止 DFN8 中断。

CRC16 为 CRC16 中断允许位。

1=允许 CRC16 中断。

0=禁止 CRC16 中断。

CRC5 为 CRC5 中断允许位。

1=允许 CRC5 中断。

0=禁止 CRC5 中断。

PIDERR 为 PIDERR 中断允许位。

1=允许 PIDERR 中断。

0=禁止 PIDERR 中断。

（9）USB 状态寄存器（STAT）

状态寄存器 STAT 能够显示 USB 模块的传输状态。当 MCU 接收到一个 TOKDNE 中断，读到的 STAT 寄存器值决定了先前端点通信的状态。状态寄存器中的值仅在 TOKDNEF 中断标志被确定时才是无效的。状态寄存器 STAT 实际上是在窗口读进一个由 USB 模块维护的状态 FIFO。当 USB 模块使用一个 BD，它会更新状态寄存器。如果另一个 USB 传输在 TOKDNE 中断处理前被执行，USB 模块将会把下一个传输的状态存储到状态 FIFO 中。因此，STAT 寄存器实际上是一个 4 字节 FIFO，它允许微控制器处理一个传输而串行接口引擎（SIE）处理下一个。清除 INTSTAT 寄存器中的 TOKDNEF 位将会引起 SIE 更新 STAT 寄存器为下一个 STAT 值。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | ENDP[3:0] | | | | IN | ODD | 0 | 0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

ENDP[3:0]为端点编号位。

0000　端点 0
0001　端点 1
0010　端点 2
0011　端点 3
0100　端点 4
0101　端点 5
0110　端点 6

IN 为进/出传输位。

1=最后 BDT 更新是为发送（进）传输而更新的。

0=最后传输是一个接收（出）数据传输。

ODD 为奇/偶传输位。

1=最后一次缓冲区描述符更新是在 BDT 的奇数存储区。

0=最后一次缓冲区描述符更新是在 BDT 的偶数存储区。

（10）USB 控制寄存器（CTL）

CTL 为 USB 模块提供了各种控制和配置信息。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | TSUSPEND | 0 | 0 | CRESUME | ODDRST | USBEN |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

TSUSPEND 为传输挂起位。

1=在一个初始化令牌收到时由 SIE 将此位置 1，SIE 已经禁止了包传送和接收。

0=允许 SIE 继续处理令牌。

CRESUME 为恢复信号位。

1=执行恢复信号，远程唤醒。

0=不执行远程唤醒。

ODDRST 为奇复位。

1=复位所有缓冲区标识符奇乒/乓位为 0，这将指定偶标识符存储区。

0=不复位。

USBEN 为 USB 使能位。

1=允许 USB 模块操作，不会影响收发器和 VREG。

0=禁用 USB 模块。

此位置 1 将会使能 USB 模块的操作。此位置 1 将引起 SIE 复位它的所有 ODD 位到 BDT。

（11）地址寄存器（ADDR）

ADDR 寄存器包含唯一的 7 位设备地址，这个地址将被识别如 USB 一样。在复位输入已经激活或者 USB 模块已经解析了 USB 复位信号以后，这个寄存器将被重置为 0x00。当固件处理一个设置地址（SET_ADDRESS）请求时，它将会改变这个值。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | ADDR6 | ADDR5 | ADDR4 | ADDR3 | ADDR2 | ADDR1 | ADDR0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

ADDR[6:0]为 USB 地址位。这 7 位定义了 USB 模块解码的 USB 地址。

（12）帧编号寄存器（FRMNUML，FRMNUMH）

帧编号寄存器包含了 11 位帧编码。帧编码寄存器要求两个 8 位寄存器来实现。低 8 位在寄存器 FRMNUML 中，而高 8 位在寄存器 FRMNUMH 中。无论何时收到一个 SOF 令牌，这两个寄存器都会更新成当前的帧编码。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | FRM7 | FRM6 | FRM5 | FRM4 | FRM3 | FRM2 | FRM1 | FRM0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FRM[7:0]为帧编码位。这些位出现在 11 位帧编码的低 8 位。

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | 0 | 0 | 0 | FRM10 | FRM9 | FRM8 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FRM[10:8]为帧编码位。这些位出现在 11 位帧编码的高 8 位。

（13）端点控制寄存器（EPCTL*n*，*n*=0～6）

每一个解析过地址的 USB 模块可用端点的控制寄存器都包含了这些端点控制位：

EPCTLDIS、EPRXEN、EPTXEN、EPHSHK。这 4 位定义了所有对任一端点的控制信息。端点 0（ENDP0）与控制管道 0 相关联，这是由 USB 的所有功能所要求的。因此，在一次 USBRST 中断接收到以后，MCU 必须设置 EPCTL0 为 0x0D。位定义如下：

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | 0 | EPCTLDIS | EPRXEN | EPTXEN | EPSTALL | EPHSHK |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

EPCTLDIS 为端点控制位。

这一位定义了一个端点是否被使能和端点方向。端点使能/方向控制如表 1-15 所示。

EPRXEN 为端点 Rx 使能位。

这一位定义了一个端点是否能够外向传送。端点使能/方向控制如表 1-15 所示。

EPTXEN 为端点 Tx 使能位。

这一位定义了一个端点是否能够内向传输。端点使能/方向控制如表 1-15 所示。

EPSTALL 为端点拖延位。

1=端点 n 被拖延。

0=端点 n 没有被拖延。

该位在端点控制寄存器中对其他位有优先权，但是仅在 EPTXEN=1 或 EPRXEN=1 时才是有效的。对这个端点的任何存取将会引起 USB 模块返回拖延握手。一旦一个端点被拖延，它需要来自主控制器的干涉。

EPHSHK 为端点握手位。

1=在一次对端点的传输期间执行了握手。

0=在一次对端点的传输期间没有执行握手。

**表 1-15　端点使能/方向控制列表**

| 位名 | | | 端点使能/方向控制 |
|---|---|---|---|
| 4 | 3 | 2 | |
| EPCTLDIS | EPRXEN | EPTXEN | |
| X | 0 | 0 | 禁用端点 t |
| X | 0 | 1 | 仅对内向（TX）传输使能端点 |
| X | 1 | 0 | 仅对外向（RX）传输使能端点 |
| 0 | 1 | 1 | 对内向、外向和初始化传输使能端点 |
| 1 | 1 | 1 | 保留 |

## 1.3 本章总结

本章首先介绍了 HCS08 单片机的基本特点，然后以 MCS08JM60 为例详细介绍了 HCS08 单片机的硬件内部结构，包括：HCS08 的 CPU、存储器、复位和中断、多功能时钟发生器、I/O 端口、KBI 模块、定时器/PWM 模块、IIC 模块、FLASH 模块、SCI 模块、SPI 模块、ADC 模块及 USB 模块。通过本章的学习，读者可以对 HCS08 单片机的硬件组成有一个大致的了解，为后续的学习打下一定的基础。

# 第 2 章

# HCS08 单片机的指令系统

HCS08 CPU 与 HC08 CPU 的目标代码和源代码完全兼容，并且由于其增加了几种寻址方式，提高了 C 语言的编译效率。本章对 HCS08 单片机的指令系统作一个具体介绍。首先介绍 HCS08 内核特点和寻址方式，然后给出 HCS08 的指令集。

## 2.1 HCS08 内核的介绍

### 2.1.1 HCS08 内核特点

HCS08 CPU 具有如下特点：

- 目标代码完全向上兼容 HC05 和 HC08 家族
- 所有寄存器和存储器映射在单块 64KB 的地址空间中
- 16 位堆栈指针（64KB 地址空间中的任意大小堆栈）
- 16 位索引寄存器（H：X）的强大寻址索引模式
- 8 位累加器（A）
- 4 种寻址模式结合的存储器到存储器数据搬移指令
- 高效的位操作指令
- 快速的 8 位乘 8 位和 16 位除以 8 位指令
- 低功耗的停止和等待模式

### 2.1.2 HCS08 CPU 寄存器

如图 2-1 所示为 5 个 CPU 寄存器：累加器（A）、索引寄存器（HX）、堆栈指针 SP、

程序计数器 PC 和条件码寄存器 CCR。CPU 寄存器不是存储器映射的一部分，它们可以被指令直接执行或被控制器直接控制。

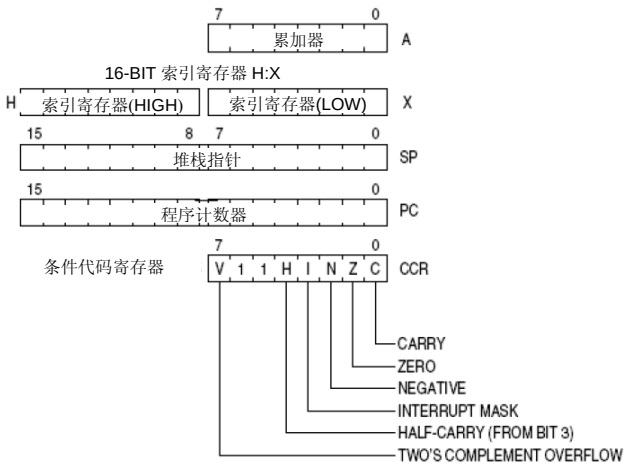



图 2-1　CPU 寄存器

### 1. 累加器 A

累加器 A 是一个通用 8 位寄存器，用来存放操作数和运算结果。累加器能从存储器加载，利用不同的寻址模式确定加载数据的地址，或者 A 中内容能被保存到存储器中，利用不同的寻址模式确定 A 中数据将要存储的地址。在执行算术、逻辑操作时，累加器首先存放一个操作数，执行完毕时累加器存放操作结果。复位时，累加器 A 中的数据不受影响。

### 2. 索引寄存器（H：X）

这个 16 位寄存器实际上分为两个 8 位寄存器（H 和 X），它们作为一个 16 位的地址指针一起工作，H 存放一个地址的高位字节，X 存放地址的低位字节。所有索引寻址模式指令利用 H：X 中全部 16 位值作为一个索引参考指针。为了与 HC05、HC08 家族兼容，一些指令只在低 8 位（X）中运行。许多指令将 X 作为第二个通用 8 位寄存器，用来存放 8 位数据值。同时，为兼容 HC05 和 HC08 家族，在复位时，H 被清零，但 X 位不受影响。

### 3. 堆栈指针 SP

堆栈指针 SP 指向下一个有效的后进先出（LIFO）堆栈位置。堆栈可以放置在包含 RAM 的 64KB 地址空间的任意位置，其大小可以为任何 RAM 中的有效值。堆栈用来保存子程序调用的返回地址和中断中 CPU 寄存器和局部变量的返回地址。AIS（立即加堆栈指针）指令对 SP 加上一个 8 位有符号立即数，这经常用于对堆栈中的局部变量分配和取消分配空间。为了与 HC05 和 HC08 家族兼容，SP 在复位中强制为 0x00FF。HCS08 程序通常在复位初始化中改变 SP 的值为片上 RAM 的最后位置（即最高地址），以便预留出直接页面 RAM（从片上寄存器的末尾到 0x00FF），用来存放程序中经常被访问的变量和用来寻址的变量。

### 4. 程序计数器 PC

程序计数器是一个 16 位的寄存器，存放取出的下一个指令或操作数的地址。在正常的程序执行期间，每一次取出指令或者操作数时，程序计数器自动加 1 指向后续的存储器位置。跳转、分支、中断和返回操作，程序计数器加载地址不是下一个后续位置。复位时，程序计数器 PC 装入复位矢量地址中的数据。一般，MC68HC08 系列中地址 0xFFFE 和 0xFFFF 中的数据是复位的入口地址，这样，复位后，程序能够从复位入口地址开始执行程序。

### 5. 条件代码寄存器 CCR

8 位条件代码寄存器存放中断屏蔽(I)和 5 个指出刚执行指令结果的标志。第 5 位(D5)和第 6 位（D6）永远为 1。下面分别介绍其余各位的含义。

（1）V（D7）——二进制补码溢出标志位

当发生一个二进制补码溢出时，CPU 置该位。有符号跳转指令 BGT、BGE、BLE 和 BLT 使用该标志位。

（2）H（D4）——半进位标志位

执行加法指令（ADD）和带进位加法指令（ADC）时，如果相加结果的第 3 位向第 4 位有进位，则 CPU 将半进位标志位置 1。该标志位对一般运算是没用的，但在二进制编码的十进制（BCD）运算中则很有用。十进制调整（DAA）指令，利用 H 和 C 的状态自动加一个修正值到先前的 ADD 或 ADC 的 BCD 运算，以修正结果为一个正确的 BCD 值。

（3）I（D3）——中断屏蔽位

1=屏蔽中断。

0=使能中断。

响应中断时，CPU 将除 H 寄存器以外的寄存器推入堆栈，置该位为 1，然后执行中断服务子程序，遇到 RTI 指令时，从栈中恢复包括 CCR 在内的各寄存器。

（4）N（D2）——负数标志位

当一个算术操作、逻辑操作或者数据处理产生一个负数结果时，则将负标志位 N 置 1。如果寄存器或存储单元的 D7 为 1，那么把寄存器或存储单元中的数据送入累加器 A 时，就会使 N 置 1；如果寄存器或存储单元的 D7 为 0，那么把寄存器或存储单元中的数据送入累加器 A 时，就会使 N 置 0。

（5）Z（D1）——零标志位

当 CPU 在进行算术操作、逻辑操作或数据处理时产生一个 0x00 或 0x0000 的结果时，零标志位被置 1，否则被清零。

（6）C（D0）——进位/借位标志位

当进行加法操作时在最高位 D7 上有进位，或当进行减法操作时 D7 需要向更高位借位时，则 CPU 将进位/借位标志位 C 置 1，否则清零该位。一些指令如位测试、移位、跳转指令等会影响该标志位。

## 2.2 寻址方式

寻址方式定义为 CPU 访问操作数和数据的方法。在 HCS08 中，所有的存储器、状态和控制寄存器、I/O 端口共享一个 64Kbyte 的线性地址空间，因此一个 16 位的二进制地址能够唯一确定任何存储器地址，这种排列意味着能够访问 RAM 中变量的指令同样能够用来访问 I/O 和控制寄存器或非易失程序空间。

一些指令用到不止一种寻址方式，如 MOVE 指令用一种寻址方式确定源操作数，用另一种寻址方式确定目的操作数。对于 BRCLR、BRSET、CBEQ 和 DBNZ 这样的指令用一种寻址方式确定测试操作数的位置，然后当测试条件为真时，用另一种寻址方式确定分支目的地址。

HCS08 的寻址方式主要有以下几种：

**1. 固定寻址方式（INH）**

固定寻址为单字节指令。在这种寻址模式中，完成指令的操作数位于 CPU 寄存器中，所以 CPU 不需要为获得操作数访问存储器。

例如：

INCA：　累加器 A 自加指令。

**2. 相对寻址方式（REL）**

相对寻址模式用于确定分支指令的目的地址。当分支条件满足时，CPU 将指令中的偏移量和程序计数器 PC 中的数据相加，得出转移的目标地址。如果分支条件不满足，则 CPU 不执行下一条指令。分支指令中的 8 位偏移量是有符号的，因此分支指令的偏移量为-127~+128。偏移量用二进制补码表示。相对寻址方式的指令有：

BCC　aaa；　如果进位标志位为零，则转移到标号 aaa 处执行。

BRA　bbb；　无条件转移到标号 bbb 处。

**3. 立即寻址方式（IMM）**

立即寻址方式在指令中直接给出操作数。这种指令是双字节指令，第 1 个字节是操作码，第 2 个字节是参与操作的立即数。立即寻址指令通常是对立即数和累加器中的数据或变址寄存器中的数据进行操作。下述指令属于 IMM 寻址方式：

LDA　#$DE；把十六进制数立即数$DE 放入累加器 A 中。

CPX　#$3A；比较 $3A 与变址寄存器 X 中的数据。

**4. 直接寻址方式（DIR）**

直接寻址指令，字长两字节。指令的第 1 个字节是操作码，第 2 个字节是操作地址的低 8 位。在直接寻址方式中，CPU 将$00 作为操作数高位字节，与指令的低字节地址直接连接，以得到期望操作数的位置的 16 位地址。其寻址范围为$0000~ $00FF。

例如：

BSET　3，　$80；将存储单元$0080 中的第 3 位置为 1。

**5. 扩展寻址方式（EXT）**

扩展寻址指令是 3 字节，可以访问存储器中的任何地址。指令的第 1 个字节是操作码，第 2 个字节是操作数的高字节地址，第 3 个字节是操作数的低字节地址。扩展寻址方式的寻址范围为$0000~ $FFFF。下述指令属于扩展寻址方式：

LDA　　$0100；将存储单元$0100 中的数值取至累加器 A。

LDHX　 $0100；将存储单元$0100 中的数值取至 H，将$0101 的数值取至 X。

与 HC08 指令系统相比，HCS08 的扩展方式中还增加了一条 CALL 指令，用于分页窗口所映射的存储器地址空间的调用，可以在 64KB 寻址空间或其他扩展页内寻址。

**6. 变址寻址方式（IX）**

（1）无偏移量变址寻址方式（IX）

无偏移量变址寻址是单字节指令。变址寄存器 HX 中的 16 位数据是操作数的地址，所以其寻址范围为$0000~$FFFF。

例如：

LDA　，X；从变址寄存器 HX 指向的存储单元中读取数据到累加器 A 中。

（2）无偏移量、后递增的变址寻址方式（IX+）

这种寻址方式与无偏移量变址寻址方式的区别在于：在本寻址方式中，操作数取出之后，多了（HX = HX + $0001）的操作。无偏移量、后递增的变址寻址方式仅用于 MOV 和 CBEQ 指令。

（3）8 位偏移量变址寻址方式（IX1）

8 位偏移量变址寻址指令是双字节指令。CPU 把变址寄存器 HX 中的 16 位数据与指令第 2 个字节的无符号 8 位偏移量相加，和即为操作数的地址。

例如：

STA　$80，X；将 A 中的数据存储至地址为（HX）+ $80 的存储单元中。

（4）8 位偏移量、后递增的变址寻址方式（IX1+）

这种寻址方式与 8 位偏移量变址寻址方式的区别在于：在本寻址方式中，在操作数取出之后，多了（HX = HX + $0001）的操作。8 位偏移量、后递增的变址寻址方式仅用于 CBEQ 指令。

例如：

CBEQ　 addr8，X+，rel；若 A=（（HX）+ addr8），则转移，（HX）+1→HX。

（5）16 位偏移量变址寻址方式（IX2）

16 位偏移量变址寻址指令是 3 字节指令，可以访问存储器中的任何地址。CPU 把变址寄存器 HX 中的 16 位数据和指令第 2、3 个字节的无符号 16 位偏移量相加，和即为操作数的地址。指令的第 2 个字节是高位地址，第 3 个字节是低位地址。例如：

STA　$200，X；将 A 中的数据写入地址为（HX）+$ 200 的存储单元中。

**7. 堆栈寻址方式（SP）**

（1）8 位偏移量堆栈寻址方式（SP1）

8 位偏移量堆栈寻址方式将指令给出的 8 位直接地址与堆栈指针 SP 相加形成操作数的地址。

例如：

LDA　2，SP；(SP)+2→A

（2）16 位偏移量堆栈寻址方式（SP2）

16 位偏移量堆栈寻址方式将指令给出的 16 位直接地址与堆栈指针 SP 相加形成操作数的地址。

例如：

LDA　$130，SP；(SP)+$130→A

# 2.3　HCS08 指令系统

CPU 所能执行的各种指令的集合称为该微处理器的指令系统。HCS08 CPU 共有 121 个保留字，加上寻址方式形成了 282 条具体指令。按指令功能将这些指令分为数据传送、算术运算、逻辑运算、位操作、移位、程序转移和控制类指令几类。

## 2.3.1　数据传送指令

数据传送指令实现 CPU 寄存器、I/O 寄存器、RAM 和 FLASH 之间的数据传送，一般情况下指令执行结果只影响 CCR 的“N”和“Z”标志位。若所传送的数据为 0，置 Z=1，否则置 Z=0；数据最高位为 1，置 N=1，否则置 N=0。

**1. 取数指令**

取数指令的功能是取出存储单元的数据存入累加器 A、X、HX 中，如表 2-1 所示。

表 2-1　取数指令的功能描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| LDA　#opr8i | 将存储单元中的数据送到累加器 A 中 | A←(M) | 0 | – | – | √ | √ | – | IMM | 2 |
| LDA　opr8a | | | | | | | | | DIR | 3 |
| LDA　opr8a | | | | | | | | | EXT | 4 |
| LDA　oprx16，X | | | | | | | | | IX2 | 4 |
| LDA　oprx8，X | | | | | | | | | IX1 | 3 |
| LDA　，X | | | | | | | | | IX | 3 |
| LDA　oprx8，SP | | | | | | | | | SP1 | 4 |
| LDA　oprx16，SP | | | | | | | | | SP2 | 5 |

续表

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| LDX　#opr8i<br>LDX　opr8a<br>LDX　opr8a<br>LDX　oprx16，X<br>LDX　oprx8，X<br>LDX　，X<br>LDX　oprx8，SP<br>LDX　oprx16，Sp | 将存储单元中的数据送到 X 寄存器中 | X←(M) | 0 | – | – | √ | √ | – | IMM<br>DIR<br>EXT<br>IX2<br>IX1<br>IX<br>SP1<br>SP2 | 2<br>3<br>4<br>4<br>3<br>3<br>4<br>5 |
| LDHX　#opr16i<br>LDHX　opr8a<br>LDHX　opr16a<br>LDHX　oprx16，X<br>LDHX　oprx8，X<br>LDHX　，X<br>LDHX　oprx8，SP | 将存储单元中的数据送到变址寄存器中 | H:X←(M:M+1) | 0 | – | – | √ | √ | – | IMM<br>DIR<br>EXT<br>IX2<br>IX1<br>IX<br>SP1 | 3<br>4<br>5<br>6<br>5<br>5<br>5 |

**2. 存储指令**

存储命令的功能是将寄存器 A、X、HX 中的数据存储到存储单元中，如表 2-2 所示。

表 2-2　存储指令的功能描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| STA　opr8a<br>STA　opr16a<br>STA　oprx16，X<br>STA　oprx8，X<br>STA　，X<br>STA　oprx8，SP<br>STA　oprx16，SP | 将累加器中的内容存储到存储单元中 | M←(A) | 0 | – | – | √ | √ | – | DIR<br>EXT<br>IX2<br>IX1<br>IX<br>SP1<br>SP2 | 3<br>4<br>4<br>4<br>2<br>4<br>5 |
| STX　opr8a<br>STX　opr8a<br>STX　oprx16，X<br>STX　oprx8，X<br>STX　，X<br>STX　oprx8，SP<br>STX　oprx16，Sp | 将 X 寄存器的数据送到存储单元中 | M←(X) | 0 | – | – | √ | √ | – | DIR<br>EXT<br>IX2<br>IX1<br>IX<br>SP1<br>SP2 | 3<br>4<br>4<br>3<br>2<br>4<br>5 |
| STHX　opr8a<br>STHX　opr8a<br>STHX　oprx8，SP | 将变址寄存器的数据送到目标地址 | M:M+1 ← (H:X) | 0 | – | – | √ | √ | – | DIR<br>EXT<br>SP1 | 4<br>5<br>5 |

### 3. 堆栈操作指令

堆栈操作指令的功能是使 A、H、X 中的数据进栈、出栈，不改变标志位。具体描述如表 2-3 所示。

表 2-3　堆栈操作指令的功能描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| PSHA | 累加器中的数据进栈 | 推(A)入栈;SP←(SP)-1(不影响标志位) | | | | | | | INH | 2 |
| PSHH | H 寄存器中的数据进栈 | 推(H)入栈；SP←(SP)-1(不影响标志位) | | | | | | | INH | 2 |
| PSHX | X 寄存器中的数据进栈 | 推(X)入栈;SP←(SP)-1(不影响标志位) | | | | | | | INH | 2 |
| PULA | 累加器中的数据出栈 | SP←(SP)+1;弹出(A)(不影响标志位) | | | | | | | INH | 3 |
| PULH | H 寄存器中的数据出栈 | SP←(SP)+1;弹出(H)(不影响标志位) | | | | | | | INH | 3 |
| PULX | X 寄存器中的数据出栈 | SP←(SP)+1;弹出(X)(不影响标志位) | | | | | | | INH | 3 |

### 4. 寄存器间数据传送指令

寄存器间数据传送指令的功能是在寄存器 A、H、HX、SP、CCR 之间进行数据传送。具体描述如表 2-4 所示。

表 2-4　寄存器间数据传送指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| TAP | 将累加器中的数据送到条件码寄存器 | CCR←(A) | √ | √ | √ | √ | √ | √ | INH | 1 |
| TAX | 将累加器中的数据送到 X 寄存器 | X←(A)(不影响标志位) | | | | | | | INH | 1 |
| TPA | 将条件码寄存器中的数据送到累加器中 | A←(CCR)(不影响标志位) | | | | | | | INH | 1 |
| TSX | 将堆栈指针送到变址寄存器 | H:X←(SP)+1(不影响标志位) | | | | | | | INH | 2 |
| TXA | 将 X 寄存在中的数据送到累加器 | A←(X)(不影响标志位) | | | | | | | INH | 1 |
| TXS | 将变址寄存器中的数据送到堆栈指针 | SP←(H:X)-1(不影响标志位) | | | | | | | INH | 2 |

### 5. 存储单元间数据传送指令

这组指令的功能是在存储单元间进行数据传送。具体描述如表 2-5 所示。

表 2-5　存储单元间数据传送指令功能描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| MOV　opr8a ,opr8a | 传送数据 | (M)目的←(M)源 | 0 | – | – | √ | √ | – | DIR/DIR | 5 |
| MOV　opr8a ,X+ | | | | | | | | | DIR/IX+ | 5 |
| MOV　#opr8i ,opr8a | | | | | | | | | IMM/DIR | 4 |
| MOV　X+,opr8a | | | | | | | | | IX+/DIR | 5 |

## 2.3.2　算术运算指令

算术运算类指令有加/减、加 1/减 1、乘/除、取反/求补、比较、测试、清零等。

**1. 加/减指令**

加/减指令的第一操作数均在 A 中，第二操作数在存储单元中，运算结果均放在 A 中。运算结果影响 V、H、N、Z、C，运算结果满足下列条件时置“1”相应的标志位，否则清零。

V　若溢出，则置位，否则清零。

H　加法，若相加结果低 3 位向高 4 位有进位，则置位，否则清零。

N　若结果的最高位为 1，相当于有符号运算中的结果为负，则置位，否则清零。

Z　若结果为 0 时，则置位，否则清零。

C　若有最高位的进位（减法为借位），则置位，否则清零。

加法有不带进位加法（ADD）和带进位加法（ADC）两种，减法有不带借位减法（SUB）和带借位减法（SBC）两种。具体描述分别如表 2-6 和表 2-7 所示。

表 2-6　加法指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| ADD　#opr8i | 不带进位加 | A←(A)+(M) | √ | √ | – | √ | √ | √ | IMM | 2 |
| ADD　opr8a | | | | | | | | | DIR | 3 |
| ADD　opr16a | | | | | | | | | EXT | 4 |
| ADD　oprx16，X | | | | | | | | | IX2 | 4 |
| ADD　oprx8，　X | | | | | | | | | IX1 | 3 |
| ADD ，X | | | | | | | | | IX | 3 |
| ADD　oprx8，SP | | | | | | | | | SP1 | 4 |
| ADD　oprx16，SP | | | | | | | | | SP2 | 5 |
| ADC　#opr8i | 带进位加 | A←(A)+(M)+(C) | √ | √ | – | √ | √ | √ | IMM | 2 |
| ADC　opr8a | | | | | | | | | DIR | 3 |
| ADC　opr16a | | | | | | | | | EXT | 4 |
| ADC　oprx16，X | | | | | | | | | IX2 | 4 |
| ADC　oprx8，　X | | | | | | | | | IX1 | 3 |
| ADC ，X | | | | | | | | | IX | 3 |
| ADC　oprx8，SP | | | | | | | | | SP1 | 4 |
| ADC　oprx16，SP | | | | | | | | | SP2 | 5 |

表 2-7　减法指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| SUB　#opr8i | 不带借位减 | A←(A)-(M) | √ | – | – | √ | √ | √ | IMM | 2 |
| SUB　opr8a | | | | | | | | | DIR | 3 |
| SUB　opr16a | | | | | | | | | EXT | 4 |
| SUB　oprx16，X | | | | | | | | | IX2 | 4 |
| SUB　oprx8，　X | | | | | | | | | IX1 | 3 |
| SUB ，X | | | | | | | | | IX | 3 |
| SUB　oprx8，SP | | | | | | | | | SP1 | 4 |
| SUB　oprx16，SP | | | | | | | | | SP2 | 5 |

续表

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| SBC #opr8i | 带借位减 | A←(A)-(M)+(C) | √ | – | – | √ | √ | √ | IMM | 2 |
| SBC opr8a | | | | | | | | | DIR | 3 |
| SBC opr16a | | | | | | | | | EXT | 4 |
| SBC oprx16，X | | | | | | | | | IX2 | 4 |
| SBC oprx8， X | | | | | | | | | IX1 | 3 |
| SBC ，X | | | | | | | | | IX | 3 |
| SBC oprx8，SP | | | | | | | | | SP1 | 4 |
| SBC oprx16，SP | | | | | | | | | SP2 | 5 |

**2. 加1/减1指令**

加 1 指令中的 INCA、INCX 是把寄存器中的数据加 1 后放回寄存器中，其他指令是把存储单元中的数据加 1 后放回原存储单元中，对标志位 V、N、Z 产生影响。减 1 指令与加 1 指令类似，只是将上述指令中的“加 1”改为“减 1”，其他一致。具体如表 2-8 所示。

表 2-8 加 1/减 1 指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| INC opr8a | 加 1 | M←(M)+1 | √ | – | – | √ | √ | – | DIR | 5 |
| INCA | | A←(A)+1 | | | | | | | INH | 1 |
| INCX | | X←(X)+1 | | | | | | | INH | 1 |
| INC oprx8， X | | M←(M)+1 | | | | | | | IX1 | 5 |
| INC ，X | | M←(M)+1 | | | | | | | IX | 4 |
| INC oprx8，SP | | M←(M)+1 | | | | | | | SP1 | 6 |
| DEC opr8a | 减 1 | M←(M)-1 | √ | – | – | √ | √ | – | DIR | 5 |
| DECA | | A←(A)-1 | | | | | | | INH | 1 |
| DECX | | X←(X)-1 | | | | | | | INH | 1 |
| DEC oprx8， X | | M←(M)-1 | | | | | | | IX1 | 5 |
| DEC ，X | | M←(M)-1 | | | | | | | IX | 4 |
| DEC oprx8，SP | | M←(M)-1 | | | | | | | SP1 | 6 |

**3. 乘/除指令**

乘法指令 MUL 将 X 与 A 中两个无符号数相乘，结果是两个字节的无符号数，其高字节放入 X 中，低字节放入 A 中，同时清零 H 和 C 标志位，不影响其他标志位。除法指令 DIV 将 H、A 中无符号 16 位数除以 X 中的 8 位无符号数，商放入 A 中，余数放入 H 中。若商大于 $FF，则将标志位 C 置 1，否则清零；若商为 0，则将标志位 Z 置 1，否则清零。不影响其他标志位。具体如表 2-9 描述。

表 2-9　乘/除指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| MUL | 乘法指令 | X:A←(X)*(A) | 0 | – | – | – | – | 0 | INH | 5 |
| DIV | 除法指令 | A←(H:A)/(X)<br>H←余数 | – | – | – | – | √ | √ | INH | 6 |

**4. 比较指令**

比较指令的实质是减法操作，如表 2-10 描述。与减法不同的是，它不需要减法的结果，不改变任何操作数，只按运算结果改变 V、N、Z、C 标志位，改变的方式与减法指令相同。

表 2-10　比较指令功能描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| CMP　#opr8i<br>CMP　opr8a<br>CMP　opr16a<br>CMP　oprx16，X<br>CMP　oprx8，X<br>CMP，X<br>CMP　oprx16，SP<br>CMP　oprx8，SP | 比较累加器中的数据与存储单元中的数据 | (A)←(M)<br>(只影响标志位) | √ | – | – | √ | √ | √ | IMM<br>DIR<br>EXT<br>IX2<br>IX1<br>IX<br>SP2<br>SP1 | 2<br>3<br>4<br>4<br>3<br>3<br>5<br>4 |
| CPX　#opr8i<br>CPX　opr8a<br>CPX　opr16a<br>CPX　oprx16，X<br>CPX　oprx8，X<br>CPX，X<br>CPX　oprx16，SP<br>CPX　oprx8，SP | 比较 X 寄存器中的数据与存储单元中的数据 | (X)←(M)<br>(只影响标志位) | √ | – | – | √ | √ | √ | IMM<br>DIR<br>EXT<br>IX2<br>IX1<br>IX<br>SP2<br>SP1 | 2<br>3<br>4<br>4<br>3<br>3<br>5<br>4 |
| CPHX　#opr16i<br>CPHX　opr8a<br>CPHX　opr16a<br>CPHX　oprx8，SP | 比较变址寄存器中的数据与存储单元中的数据 | (H:X)←(M:M+$0001)<br>（只影响标志位） | √ | – | – | √ | √ | √ | IMM<br>DIR<br>EXT<br>SP1 | 3<br>5<br>6<br>6 |

**5. 取反/求补指令**

取反指令中的 COMA、COMX 是把寄存器中的数据取反后放回到寄存器中，如表 2-11 所示；其他指令是把存储单元中的数据取反后放回到原存储单元中。求补指令中的 NEGA、NEGX 是把寄存器中的数据求补后放回到寄存器中，其他指令是把存储单元中的数据求补后放回到原存储单元中。

表 2-11　取反/求补指令

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| COM　opr8a | 取反 | M←(M)=$FF-(M) | 0 | – | – | √ | √ | 1 | DIR | 5 |
| COMA | | A←(A)=$FF-(A) | | | | | | | INH | 1 |
| COMX | | X←(X)=$FF-(X) | | | | | | | INH | 1 |
| COM　oprx8，X | | M←(M)=$FF-(M) | | | | | | | IX1 | 5 |
| COM ，X | | M←(M)=$FF-(M) | | | | | | | IX | 4 |
| COM　oprx8，SP | | M←(M)=$FF-(M) | | | | | | | SP1 | 6 |
| NEG　opr8a | 求补 | M←-(M)=$00-(M) | 0 | – | – | √ | √ | 1 | DIR | 5 |
| NEGA | | A←-(A)=$00-(A) | | | | | | | INH | 1 |
| NEGX | | X←-(X)=$00-(X) | | | | | | | INH | 1 |
| NEG　oprx8，X | | M←-(M)=$00-(M) | | | | | | | IX1 | 5 |
| NEG ，X | | M←-(M)=$00-(M) | | | | | | | IX | 4 |
| NEG　oprx8，SP | | M←-(M)=$00-(M) | | | | | | | SP1 | 6 |

**6. 测试指令**

测试指令中的 TSTA、TSTX 是将寄存器中的数据与零比较（实际上是执行减 0 的操作），其他指令是把存储单元中的数据与零比较。测试指令执行后并不改变操作数本身，清零 C 标志位，根据结果改变 N、Z 标志位。具体如表 2-12 描述。

表 2-12　测试指令功能描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| TST　opr8a | 符号及零标志的测试 | (A)- =$00 or (X)- =$00 or (M)- =$00 | 0 | – | – | √ | √ | – | DIR | 5 |
| TSTA | | | | | | | | | INH | 1 |
| TSTX | | | | | | | | | INH | 1 |
| TST　oprx8，X | | | | | | | | | IX1 | 5 |
| TST ，X | | | | | | | | | IX | 4 |
| TST　oprx8，SP | | | | | | | | | SP1 | 6 |

**7. 清零指令**

清零指令中的 CLRA、CLRX、CLRH 是将寄存器清零，其他指令是将存储单元清零，同时清零 V、N 标志位，置 Z 标志位，不改变 H、I、C 标志位。具体如表 2-13 所示。

表 2-13　清零指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| CLR　opr8a | 清零 | M←$00 | 0 | – | – | 0 | 1 | – | DIR | 5 |
| CLRA | | A←$00 | | | | | | | INH | 1 |
| CLRX | | X←$00 | | | | | | | INH | 1 |
| CLRH | | H←$00 | | | | | | | INH | 1 |
| CLR　oprx8，X | | M←$00 | | | | | | | IX1 | 5 |
| CLR ，X | | M←$00 | | | | | | | IX | 4 |
| CLR　oprx8，SP | | M←$00 | | | | | | | SP1 | 6 |

## 2.3.3　逻辑运算指令

逻辑运算指令完成逻辑与、或、异或操作，如表 2-14 所示。这些指令把累加器 A 中的数据与存储单元中的数据进行运算后，结果再送到累加器中，同时清零 V 标志位。

表 2-14　逻辑运算指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| AND　#opr8i | 逻辑与 | A←(A)&(M) | 0 | – | – | √ | √ | – | IMM | 2 |
| AND　opr8a | | | | | | | | | DIR | 3 |
| AND　opr16a | | | | | | | | | EXT | 4 |
| AND　oprx16，X | | | | | | | | | IX2 | 4 |
| AND　oprx8，X | | | | | | | | | IX1 | 3 |
| AND，X | | | | | | | | | IX | 3 |
| AND　oprx16，SP | | | | | | | | | SP2 | 5 |
| AND　oprx8，SP | | | | | | | | | SP1 | 4 |
| ORA　#opr8i | 逻辑或 | A←(A)\|(M) | 0 | – | – | √ | √ | – | IMM | 2 |
| ORA　opr8a | | | | | | | | | DIR | 3 |
| ORA　opr16a | | | | | | | | | EXT | 4 |
| ORA　oprx16，X | | | | | | | | | IX2 | 4 |
| ORA　oprx8，X | | | | | | | | | IX1 | 3 |
| ORA，X | | | | | | | | | IX | 3 |
| ORA　oprx16，SP | | | | | | | | | SP2 | 5 |
| ORA　oprx8，SP | | | | | | | | | SP1 | 4 |
| EOR　#opr8i | 逻辑异或 | A←(A)♀(M) | 0 | – | – | √ | √ | – | IMM | 2 |
| EOR　opr8a | | | | | | | | | DIR | 3 |
| EOR　opr16a | | | | | | | | | EXT | 4 |
| EOR　oprx16，X | | | | | | | | | IX2 | 4 |
| EOR　oprx8，X | | | | | | | | | IX1 | 3 |
| EOR，X | | | | | | | | | IX | 3 |
| EOR　oprx16，SP | | | | | | | | | SP2 | 5 |
| EOR　oprx8，SP | | | | | | | | | SP1 | 4 |

## 2.3.4　位操作指令

位操作类指令主要完成位测试、位置 1、位清零等操作。如表 2-15 所示，位测试指令 BIT 与 AND 的区别是把累加器中的数据与存储单元中的数据进行与运算后，结果不再送到累加器中，即它不改变参与运算的操作数。BCLR 和 BSET 两条指令可对存储空间起始的 256 个单元的任一可读写位执行清零或置位操作。指令中 *n*=7~0，指明一个字节的第几位，不影响任何标志位。

表 2-15　位操作指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| BIT　#opr8i | 位测试 | (A)&(M) | 0 | - | - | √ | √ | - | IMM | 2 |
| BIT　opr8a | | | | | | | | | DIR | 3 |
| BIT　opr16a | | | | | | | | | EXT | 4 |
| BIT　oprx16，X | | | | | | | | | IX2 | 4 |
| BIT　oprx8，X | | | | | | | | | IX1 | 3 |
| BIT ，X | | | | | | | | | IX | 3 |
| BIT　oprx16，SP | | | | | | | | | SP2 | 5 |
| BIT　oprx8，SP | | | | | | | | | SP1 | 4 |
| BCLR　n，opr8a | 清零存储单元的第 *n* 位 | Mn←0(不影响标志位) | | | | | | | DIR | 5 |
| BSET　n，opr8a | 置位存储单元的第 *n* 位 | Mn←1(不影响标志位) | | | | | | | DIR | 5 |
| CLC | 清零进位标志位 | C←0 | | | | | | | INH | 1 |
| CLI | 清零中断屏蔽标志位 | I←0 | | | | | | | INH | 1 |
| SEC | 置位进位标志位 | C←1 | | | | | | | INH | 1 |
| SEI | 置位中断屏蔽标志位 | I←1 | | | | | | | INH | 1 |

## 2.3.5　移位类指令

移位类指令包括算术左移、算术右移、逻辑左移、逻辑右移、循环移位等，如表 2-16、表 2-17 所示。

表 2-16　移位类指令描述 1

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| ASL　opr8a | 算术左移 | | √ | – | – | √ | √ | √ | DIR | 5 |
| ASLA | | | | | | | | | INH | 1 |
| ASLX | | | | | | | | | INH | 1 |
| ASL　oprx8，X | | | | | | | | | IX1 | 5 |
| ASL ，X | | | | | | | | | IX | 4 |
| ASL　oprx8，SP | | | | | | | | | SP1 | 6 |
| ASL　opr8a | 算术右移 | | √ | – | – | √ | √ | √ | DIR | 5 |
| ASLA | | | | | | | | | INH | 1 |
| ASLX | | | | | | | | | INH | 1 |
| ASL　oprx8，X | | | | | | | | | IX1 | 5 |
| ASL ，X | | | | | | | | | IX | 4 |
| ASL　oprx8，SP | | | | | | | | | SP1 | 6 |

表 2-17　移位类指令描述 2

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| LSL　opr8a<br>LSLA<br>LSLX<br>LSL　oprx8，　X<br>LSL ，X<br>LSL　oprx8，SP | 逻辑左移 | | √ | – | – | √ | √ | √ | DIR<br>INH<br>INH<br>IX1<br>IX<br>SP1 | 5<br>1<br>1<br>5<br>4<br>6 |
| LSR　opr8a<br>LSRA<br>LSRX<br>LSR　oprx8，　X<br>LSR ，X<br>LSR　oprx8，SP | 逻辑右移 | | √ | – | – | √ | √ | √ | DIR<br>INH<br>INH<br>IX1<br>IX<br>SP1 | 5<br>1<br>1<br>5<br>4<br>6 |
| ROL　opr8a<br>ROLA<br>ROLX<br>ROL　oprx8，　X<br>ROL ，X<br>ROL　oprx8，SP | 带进位标志位的循环左移 | | √ | – | – | √ | √ | √ | DIR<br>INH<br>INH<br>IX1<br>IX<br>SP1 | 5<br>1<br>1<br>5<br>4<br>6 |
| ROR　opr8a<br>RORA<br>RORX<br>ROR　oprx8，　X<br>ROR ，X<br>ROR　oprx8，SP | 带进位标志位的循环右移 | | √ | – | – | √ | √ | √ | DIR<br>INH<br>INH<br>IX1<br>IX<br>SP1 | 5<br>1<br>1<br>5<br>4<br>6 |

## 2.3.6　程序转移和控制类指令

程序转移和控制类指令包括无条件转移指令、条件转移指令、调用子程序指令、返回指令和控制指令。

**1. 无条件转移指令**

无条件转移指令又分为相对转移指令和绝对转移指令。绝对转移指令将所寻址的两个单元内容作为地址送到 PC，使 CPU 转到该地址开始执行程序，不影响标志位。具体如表 2-18 所示。

表 2-18　无条件转移指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| BRA　rel | 无条件相对转移指令 | PC←(PC)+2+rel<br>（不影响标志位） | | | | | | | REL | 3 |
| JMP　opr8a<br>JMP　opr16a<br>JMP　oprx8，X<br>JMP　oprx16，X<br>JMP，X | 无条件绝对转移指令 | PC←跳转地址<br>（不影响标志位） | | | | | | | DIR<br>EXT<br>IX2<br>IX1<br>IX | 3<br>4<br>4<br>3<br>3 |

**2. 条件转移指令**

条件转移指令又可分为：测试状态标志条件转移指令、测试 IRQ 条件转移指令、位测试转移指令、减 1 不为零转移和比较相等转移指令。

（1）测试状态条件转移指令

这类指令根据上一条指令的执行结果测试 CCR 的标志位状态。若条件满足，执行相对转移指令；若条件不满足，则顺序执行下一条指令，不影响标志位。具体如表 2-19 所示。

表 2-19　测试状态条件转移指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| BCC　rel | 进位标志位为 0 则转移 | PC←(PC)+2+rel?(C)=0 | | | | | | | REL | 3 |
| BCS　rel | 进位标志位为 1 则转移 | PC←(PC)+2+rel?(C)=1 | | | | | | | REL | 3 |
| BEQ　rel | 相等则转移 | PC←(PC)+2+rel?(Z)=1 | | | | | | | REL | 3 |
| BNE　rel | 不等则转移 | PC←(PC)+2+rel?(Z)=0 | | | | | | | REL | 3 |
| BHI　rel | 大于则转移 | PC←(PC)+2+rel?(C)\|(Z)=0 | | | | | | | REL | 3 |
| BLS　rel | 小于或等于则转移 | PC←(PC)+2+rel?(C)\|(Z)=1 | | | | | | | REL | 3 |
| BHCC rel | 半进位标志位为 0 则转移 | PC←(PC)+2+rel?(H)=0 | | | | | | | REL | 3 |
| BHCS rel | 半进位标志位为 1 则转移 | PC←(PC)+2+rel?(H)=1 | | | | | | | REL | 3 |
| BMI　rel | 负值则转移 | PC←(PC)+2+rel?(N)=1 | | | | | | | REL | 3 |
| BPL　rel | 正值则转移 | PC←(PC)+2+rel?(N)=0 | | | | | | | REL | 3 |
| BMC　rel | 中断屏蔽标志位为 0 则转移 | PC←(PC)+2+rel?(I)=0 | | | | | | | REL | 3 |
| BMS　rel | 中断屏蔽标志位为 1 则转移 | PC←(PC)+2+rel?(I)=1 | | | | | | | REL | 3 |
| BLO　rel | 小于则转移 | PC←(PC)+2+rel?(C)=1 | | | | | | | REL | 3 |
| BHS　rel | 大于或等于则转移 | PC←(PC)+2+rel?(C)=0 | | | | | | | REL | 3 |
| BGT　rel | 大于则转移（有符号数） | PC←(PC)+2+rel?(Z)\|(N⊕+V)=0 | | | | | | | REL | 3 |
| BLE　rel | 小于或等于则转移（有符号数） | PC←(PC)+2+rel?(Z)\|(N⊕+V)=1 | | | | | | | REL | 3 |
| BGE　rel | 大于或等于则转移（有符号数） | PC←(PC)+2+rel?(N⊕+V)=0 | | | | | | | REL | 3 |
| BLT　rel | 小于则转移（有符号数） | PC←(PC)+2+rel?(N⊕+V)=1 | | | | | | | REL | 3 |

（2）测试 IRQ 条件转移指令

具体如表 2-20 所示，测试 IRQ 条件转移指令负责电平的高低转移操作。

表 2-20　测试 IRQ 条件转移指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| BIH　rel | IRQ 输入高电平则转移 | PC←(PC)+2+rel?IRQ =1 | | | | | | | REL | 3 |
| BIL　rel | IRQ 输入低电平则转移 | PC←(PC)+2+rel?IRQ=0 | | | | | | | REL | 3 |

（3）位测试转移指令

测试所寻址单元相应位状态，若满足条件则发生转移，否则顺序执行下一条指令。只能对零页的 RAM 或 I/O 寄存器的位测试转移，不影响标志位。具体如表 2-21 所示。

表 2-21　位测试转移指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| BRCLR n，opr8a，rel | 存储单元第 *n* 位为 0 则转移 | PC←(PC)+3+rel?Mn=0 | | | | | | | REL | 3 |
| BRSET n，opr8a，rel | 存储单元第 *n* 位为 1 则转移 | PC←(PC)+3+rel?Mn=1 | | | | | | | REL | 3 |

（4）减 1 不为零转移指令

这是兼有减 1 指令和不等于 0 转移指令功能的条件转移指令，不影响标志位。具体如表 2-22 所示。

表 2-22　减 1 不为零转移指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| DBNZ　opr8a，rel | 减去 1 后结果不为 0 则转移 | PC←(PC)+3+rel?(result)/=0 | | | | | | | DIR | 7 |
| DBNZ　rel | | PC←(PC)+2+rel?(result)/=0 | | | | | | | INH | 4 |
| DBNZX rel | | PC←(PC)+2+rel?(result)/=0 | | | | | | | INH | 4 |
| DBNZ　oprx8，X，rel | | PC←(PC)+3+rel?(result)/=0 | | | | | | | IX1 | 7 |
| DBNZ　X，rel | | PC←(PC)+2+rel?(result)/=0 | | | | | | | IX | 6 |
| DBNZ　oprx8，SP，rel | | PC←(PC)+4+rel?(result)/=0 | | | | | | | SP1 | 8 |

（5）比较相等转移指令

累加器 A 或 X 和所寻址单元内容相比较，若相等，则执行相对转移指令；若不相等，顺序执行下一条指令。不影响标志位。具体如表 2-23 所示。

表 2-23　比较相等转移指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| CBEQ　opr8a，rel | 比较，相等则转移 | PC←(PC)+3+rel?(A)-（M）=$00 | | | | | | | DIR | 5 |
| CBEQA　#opr8i,rel | | PC←(PC)+3+rel?(A)-（M）=$00 | | | | | | | IMM | 4 |
| CBEQX　#opr8i,rel | | PC←(PC)+3+rel?(A)-（M）=$00 | | | | | | | IMM | 4 |
| CBEQ　oprx8，X，rel | | PC←(PC)+3+rel?(A)-（M）=$00 | | | | | | | IX1+ | 5 |
| CBEQ　X，rel | | PC←(PC)+2+rel?(A)-（M）=$00 | | | | | | | IX+ | 5 |
| CBEQ　oprx8，SP，rel | | PC←(PC)+4+rel?(A)-（M）=$00 | | | | | | | SP1 | 6 |

### 3. 调用子程序指令

（1）相对调用子程序指令

该调用子程序指令功能和相对转移指令类似，只是增加了 PC 进栈操作，也不影响标志位。具体如表 2-24 所示。

表 2-24　相对调用子程序指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| BSR　rel | 子程序相对调用指令 | PC←(PC)+2<br>Push(PCL):SP←SP-1<br>Push(PCH):SP←SP-1<br>PC←(PC)+rel | | | | | | | REL | 4 |

（2）绝对调用子程序指令

该调用子程序指令功能和绝对转移指令类似，只是增加了 PC 进栈操作，也不影响标志位。具体如表 2-25 所示。

表 2-25　绝对调用子程序指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| JSR　opr8a | 子程序绝对调用指令 | PC←(PC)+n(n=1,2 or 3)<br>Push(PCL):SP←SP-1<br>Push(PCH):SP←SP-1<br>PC←子程序入口<br>（不影响标志位） | | | | | | | DIR | 5 |
| JSR　opr16a | | | | | | | | | EXT | 6 |
| JSR　oprx16,X | | | | | | | | | IX2 | 6 |
| JSR　oprx8,X | | | | | | | | | IX1 | 5 |
| JSR　,X | | | | | | | | | IX | 5 |

### 4. 返回指令

（1）从子程序返回指令

该条指令的功能是 PC 退栈返回主程序。一般子程序必须以 RIS 结束。具体如表 2-26 所示。

表 2-26　从子程序返回指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| RTS | 子程序返回指令 | SP←SP+1：Pull(PCH)<br>SP←SP+1：Pull(PCL) | | | | | | | INH | 6 |

（2）从中断返回指令

该条指令的功能和 RTS 相似，只是多了 CPU 寄存器 CCR、A、X 的退栈操作，使 CPU 从原来被中断的地方继续执行程序。中断服务程序必须以 RTI 指令结束。具体如表 2-27 所示。

表 2-27　从中断返回指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| RTS | 中断返回指令 | SP←SP+1：Pull(CCR)<br>SP←SP+1：Pull(A)<br>SP←SP+1：Pull(X)<br>SP←SP+1：Pull(PCH)<br>SP←SP+1：Pull(PCL) | | | | | | | INH | 9 |

**5. 控制指令**

（1）软件中断指令

CPU 执行软件中断 SWI 指令，产生不可屏蔽的软件中断，主要用于设计开发工具的监控程序。具体如表 2-28 所示。

表 2-28　软件中断指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| SWI | 软件中断指令 | PC←PC+1：Push(PCL)<br>SP←SP-1：Push(PCH)<br>SP←SP-1：Push(X)<br>SP←SP-1：Push(A)<br>SP←SP-1：Push(CCR)<br>SP←SP-1：I—1<br>PCH←中断向量高字节<br>PCL←中断向量低字节 | | | | | | | INH | 11 |

（2）堆栈指针复位指令

堆栈指针复位 RSP 指令使 SPL 值为$FF，而 SPH 值保持不变，不影响标志。具体如表 2-29 所示。

表 2-29　堆栈指针复位指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| RSP | 堆栈指针指令 | SPL←$FF | | | | | | | INH | 1 |

（3）节电方式指令

节电方式指令具体如表 2-30。

表 2-30　节电方式指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| STOP | 进入停止模式 | I←0:CPU 时钟关闭 | | | | | | | INH | 2 |
| WAIT | 进入等待模式 | I←0:CPU 时钟关闭 | | | | | | | INH | 2 |

（4）空操作指令

空操作指令本身不执行任何操作，用在延时等程序中，以调节程序执行的时间。具体如表 2-31 所示。

表 2-31　空操作指令描述

| 指令 | 操作 | 功能描述 | 条件代码寄存器 | | | | | | 寻址方式 | 周期 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | V | H | I | N | Z | C | | |
| NOP | 空操作 | （不影响标志位） | | | | | | | INH | 1 |
| BRN　rel | 不转移（空操作） | 占 3 个总线周期 | | | | | | | REL | 3 |

## 2.4　本章总结

本章主要对 HCS08 单片机的指令系统作了详细介绍。首先介绍了 HCS08 内核的特点和寻址方式，然后按指令功能，介绍了数据传送、算术运算、逻辑运算、位操作、移位、程序转移和控制类指令 6 类的指令集。为了描述直观，全部结合表格来阐述，读者通过学习，可以对 HCS08 单片机的指令系统有一个全面的掌握。

# 第 3 章

# HCS08 单片机的 C 语言程序设计

本章将介绍 HCS08 单片机的 C 语言程序设计知识。首先介绍基于单片机的 C 语言程序设计基础知识，然后介绍 HCS08 C 语言的扩展语法。C 语言是一种结构化语言，程序有规范的结构，可分为不同的函数，使得程序结构化；同时程序更具可读性，具有很高的编程及程序调试效率；并且程序易于做到模块化，便于移植。因此 C 语言被广大编程者使用。

## 3.1 单片机 C 语言设计基础

本节重点围绕着数据结构、程序结构、函数与控制语句进行介绍。

### 3.1.1 数据结构

具有一定格式的数字或数值叫做数据。数据是计算机操作的对象，无论使用何种语言、算法进行程序设计，最终在计算机中运行的只有数据流。数据的不同格式称为数据类型。数据按一定的数据类型进行的排列、组合、架构称为数据结构。数据结构是程序的基础，开始 C 语言程序的学习之前必须先掌握它所支持的各类数据结构。

#### 3.1.1.1 数据类型

C 语言的数据类型包括：基本类型、构造类型、指针类型及空类型。其中，基本类型包括位（bit）、字符（char）、整型（int）、短整型（short）、长整型（long）、浮点型（float）及双精度浮点型（double）；构造类型包括数组（array）、结构体（struct）、共用体（union）及枚举类型（enum）。

**1. 字符类型char**

char 类型的长度是 1 个字节，通常用于定义处理字符数据的变量或常量。它分为无符号字符类型 unsigned char 和有符号字符类型 signed char，默认值为 signed char 类型。unsigned char 类型用字节中的所有位表示数值，可以表达的数值范围是 0～255。signed char 类型用字节中最高位表示数据的符号，“0”表示正数，“1”表示负数，负数用补码表示，可以表达的数值范围是–128～+127。unsigned char 常用于处理 ASCII 码字符或小于等于 255 的整型数。

**2. 整型int**

int 类型的长度为 2 个字节，用于存放一个双字节数据，分有符号整型 signed int 和无符号整型 unsigned int，默认值为 signed int 类型。signed int 表示的数值范围是-32 768～+ 32 767，字节中最高位表示数据的符号，“0”表示正数，“1”表示负数，负数用补码表示。unsigned int 表示的数值范围是 0～65 535。

**3. 长整型long**

long 类型的长度为 4 个字节，用于存放一个 4 字节数据，分有符号长整型 signed long 和无符号长整型 unsigned long，默认值为 signed long 类型。signed long 表示的数值范围是 –2 147 483 648～+2 147 483 647，字节中最高位表示数据的符号，“0”表示正数，“1”表示负数，负数用补码表示。unsigned long 表示的数值范围是 0～4 294 967 295。

**4. 浮点型float**

float 类型的长度为 32 位，占用 4 个字节，许多复杂的数学表达式都采用浮点数据类型。它用符号位表示数的符号，用阶码与尾数表示数的大小。用它们进行任何数学运算都需要使用由编译器决定的各种不同效率等级的库函数。

符号位是最高位，尾数为最低的 23 位，内存中按字节存储如下：

| 地址 | +0 | +1 | +2 | +3 |
|---|---|---|---|---|
| 内容 | WWWW WWWW | WWWW WWWW | EWWW WWWW | SEEE EEEE |

其中 S 是符号位，“0”表示正值，“1”表示负值；E 是阶码（在两个字节中），偏移值为 127；W 是 23 位尾数，最高位为“1”。

**5. 指针型***

*（星号）类型本身就是一个变量，在这个变量中存放指向另一个数据的地址。这个指针变量要占据一定的内存单元，对于不同的处理器其长度也不尽相同。

**6. 16位特殊功能寄存器sfr16**

sfr16 类型占用两个内存单元，值域为 0～65 535。sfr16 和 sfr 一样用于操作特殊功能的寄存器，所不同的是它用于操作占两个字节的寄存器，如定时器 T0 和 T1。

需要说明的是：C 语言是大小写敏感的一种高级语言，如果要定义一个定时器 1，可

以写做“Timer1”，如果程序中有“TIMER1”，那么这两个定义是完全不同的标识符。标识符由字符串、数字和下画线等组成，注意第一个字符必须是字母或下画线，如“1Timer”是错误的，编译时便会有错误提示。有些编译系统专用的标识符是以下画线开头的，所以一般不要以下画线开头命名标识符。标识符在命名时应当首先含义清晰，再求简单，这样有助于阅读理解程序。

#### 3.1.1.2　变量与常量

C 语言中的数据有常量与变量之分。在程序运行过程中值不能改变的量称为常量，而变量是在程序运行过程中可以不断变化的量。

变量的定义可以使用所有编译器支持的数据类型，而常量的数据类型只有位、整型、浮点型、字符型和字符串型。

位型常量是一位二进制值。

整型常量可以表示为十进制，如：17、0、–66 等；若表示为十六进制则以 0x 开头，如：0x2A、–0x5D 等。长整型就在数字后面加字母 L，如：104L、034L、0xF340 等。

浮点型常量可分为十进制和指数表示形式。十进制由数字和小数点组成，例如：0.88888、3345.345、0.0 等；指数表示形式为：[±]数字[.数字]e[±]数字，方括号“[ ]”中的内容为可选项，其中的内容根据具体情况可有可无，但其余部分必须有，如 125e3、7e9、-3.0e-3 等。

字符型常量是单引号内的字符，如‘a’，‘d’等，不可以显示的控制字符可以在该字符前面加一个反斜杠“\”组成专用转义字符。常用转义字符如表 3-1 所示。

**表 3-1　常用转义字符**

| 转 义 字 符 | 含　义 | 16 进制 ASCII 码 |
|---|---|---|
| \0 | 空字符（NULL） | 00H |
| \b | 退格符（BS） | 08H |
| \t | 水平制表符（HT） | 09H |
| \n | 换行符（LF） | 0AH |
| \f | 换页符（FF） | 0CH |
| \r | 回车符（CR） | 0DH |
| \” | 双引号 | 22H |
| \’ | 单引号 | 27H |
| \\ | 反斜杠 | 5CH |

字符串型常量由双引号内的字符组成，如“Error”、“OK”等。当双引号内没有字符时，为空字符串。在使用特殊字符时同样要使用转义字符，如双引号。在 C 语言中字符串常量是作为字符类型数组来处理的，在存储字符串时系统会在字符串尾部加上转义字符\0，以作为该字符串的结束符。字符串常量“A”和字符常量‘A’是不同的，前者在存储时多占用一个字节用以存储结束符\0。

常量可用在不必改变值的场合，如固定的数据表、字库等。用标识符代表的常量称为

符号常量。可以通过宏定义 define 来实现用标识符号代表常量。例如：

```
#define CONST 100        // 符号常量 CONST 值为 100
```

在程序中遇到 CONST 编译时自动用 100 替换。

使用符号常量的好处是：

①含义清楚。在单片机程序中，常有一些量是具有特定含义的，如某单片机系统扩展了一些外部芯片，每一块芯片的地址即可用符号常量定义，如：

```
#define PORTA 0x7fff
#define PORTB 0x7ffe
```

程序中可以用 PORTA、PORTB 来对端口进行操作，而不必写 0x7fff、0x7ffe。显然，这两个符号比两个数字更能令人明白其含义。给符号常量起名字时，尽量要做到“见名知意”。

②在需要改变一个常量时能做到“一改全改”。如果由于某种原因，端口的地址发生了变化（如修改了硬件），由 0x7fff 改成了 0x3fff，那么只要将宏定义语句稍作改动即可，如下所示：

```
#define PORTA 0x3fff
```

这样不仅方便，而且能避免出错。如果不用符号常量，要在成百上千行程序中把所有表示端口地址的 0x7fff 找出来并修改可不是件容易的事。

变量主要由两部分构成：一个是变量名，一个是变量值。每个变量都有一个变量名，在内存中占据一定的存储单元（地址），并在该内存单元中存放该变量的值。要在程序中使用变量必须先用标识符作为变量名，并指出所用的数据类型和存储模式，这样编译系统才能为变量分配相应的存储空间。在 C 语言中，要求对所有用到的变量进行强制定义，也就是“先定义，后使用”。

定义一个变量的格式如下：

[存储种类] 数据类型 [存储器类型] 变量名表

在定义格式中除了数据类型和变量名表是必要的外，其他都是可选项。存储种类有 4 种：自动（auto），外部（extern），静态（static）和寄存器（register）。默认类型为自动（auto）。存储器共有 data、bdata、idata、pdata、xdata 及 code 6 种类型，如表 3-2 所示。

**表 3-2　存储器类型说明**

| 存储器类型 | 说　明 |
|---|---|
| data | 直接访问数据内部存储器，速度最快 |
| bdata | 可位寻址内部数据存储器，允许位与字节混合访问 |
| idata | 间接访问内部数据存储器，允许访问全部内部地址 |
| pdata | 分页访问外部数据存储器，用 MOVX @Ri 指令访问 |
| xdata | 外部数据存储器，用 MOVX @DPTR 指令访问 |
| code | 程序存储器，用 MOVC @A+DPTR 指令访问 |

如果省略存储器类型，系统则会按编译模式 SMALL、COMPACT 或 LARGE 所规定的默认存储器类型去指定变量的存储区域。变量的存储种类与存储器类型是完全无关的。

SMALL 存储模式将所有变量都放在单片机的内部 RAM 区，这使得访问数据非常快，但 SMALL 存储模式的地址空间受限。在写小型应用程序时，变量和数据放在 data 内部数据存储器中是很好的，因为访问速度快，但在较大的应用程序中 data 区最好只存放小的变量、数据或常用的变量（如循环计数、数据索引），而大的数据则放置在别的存储区域中。

COMPACT 存储模式可以使用一页（256B）外部扩展 RAM。

LARGE 存储模式可以使用全部的外部扩展 RAM，外部数据区最多可有 64KB，这要求用 DPTR 数据指针访问数据。

下面再来仔细看看单片机 C 语言程序中 sfr、sfr16 和 sbit 定义变量的方法。

sfr 和 sfr16 可以直接对单片机的特殊寄存器进行定义，定义方法如下：

```
sfr 特殊功能寄存器名 = 特殊功能寄存器地址常数;
sfr16 特殊功能寄存器名 = 特殊功能寄存器地址常数;
```

sfr 关键字后面是一个要定义的名字，可任意选取，但要符合标识符的命名规则。名字最好有一定的含义，如 P1 口可以用 P1 为名，这样程序具有可读性。等号后面必须是常数，不允许有带运算符的表达式，而且该常数必须在特殊功能寄存器的地址范围之内（80H～FFH）。

sfr 用来定义 8 位的特殊功能寄存器，而 sfr16 则用来定义 16 位特殊功能寄存器，且该 16 位必须是低位在低字节上，高位在紧随的高字节上。如 8052 的 T2 定时器可以定义为：

```
sfr16 T2 = 0xCC;          // 定义 8052 定时器 2，地址为 T2L=CCH，T2H=CDH
```

此定义方法不能用于定时器 0 或 1。

sbit 用于定义可位寻址的对象，如访问特殊功能寄存器中的某位。这种应用是经常需要的，比如要访问 P1 口中的第 2 个引脚 P1.1，可以按照以下的方法定义：

（1）sbit 位变量名 ＝ 位地址

```
sbit P1_1 = 0x91;
```

将位的绝对地址赋给位变量。同 sfr 一样，sbit 的位地址必须位于 80H 和 FFH 之间。

（2）sbit 位变量名 ＝ 特殊功能寄存器名^位位置

```
sfr P1 = 0x90;            //先定义一个特殊功能寄存器名
sbit P1_1 = P1 ^ 1;       //再指定位变量名所在的位置
```

当可寻址位位于特殊功能寄存器中时可采用此种方法。

（3）sbit 位变量名 ＝ 字节地址^位位置

```
sbit P1_1 = 0x90 ^ 1;
```

这种方法其实和（2）是一样的，只是把特殊功能寄存器的位地址直接用常数表示。

初学者容易混淆符号常量与变量，它们的区别在于：符号常量的值在其作用域中，不能改变也不能用等号赋值。

举例如下：

```
#define CONST 100
main()
{
    int var;
    int result;
    var = 10;
    result = var * CONST;
    printf（“result=%d\n”,result）;
}
```

程序运行结果：

```
result = 1000
```

程序的开头#define CONST 100 定义了一个符号常量 CONST，其值为 100。这样在后面的程序中，凡是出现 CONST 的地方，都代表常量 100。程序中的 var 和 result 都是变量。

#### 3.1.1.3 数组

字符型 char、整型 int、浮点型 float 等都属于基本数据类型。除此之外，C 语言还提供一些扩展数据类型，它们都是由基本数据类型构造而成的，因此称为构造数据类型。这些按照一定规则构造而成的数据类型包括：数组、指针、结构、共用体及枚举。

在程序设计中，为了处理方便，需要把具有相同类型的若干变量按有序的形式组织起来，这些按序排列的同类数据元素的集合称为数组。一个数组可以分解为多个数组元素，这些数组元素可以是基本数据类型或构造类型。因此按数组元素的类型不同，数组又可分为数值数组、字符数组、指针数组、结构数组等各种类别。数组有一维、二维和多维数组之分。

**1. 一维数组**

一维数组说明的一般形式为：

类型说明符 数组名 [常量表达式]；

其中，类型说明符是任意一种基本数据类型或构造数据类型。数组名是用户定义的数组标识符。方括号中的常量表达式表示数据元素的个数，也称为数组的长度。

例如：

```
int a[5];           // 说明整型数组 a，有 5 个元素。
float b[10],c[20];  // 说明实型数组 b，有 10 个元素，实型数组 c，有 20 个元素。
char ch[30];        // 说明字符数组 ch，有 30 个元素。
```

对于数组类型说明应注意以下几点：

（1）数组的类型实际上是指数组元素的取值类型。对于同一个数组，其所有元素的数据类型都是相同的。

（2）数组名的书写规则应符合标识符的书写规定。

（3）数组名不能与其他变量名相同，例如以下程序是错误的：

```
void main()
{
    int a;
    float a[10];   // 数组 a 与变量 a 同名
    ……
}
```

（4）方括号中常量表达式表示数组元素的个数，如 a[5]表示数组 a 有 5 个元素。但是其下标从 0 开始计算，因此 5 个元素分别为 a[0]、a[1]、a[2]、a[3]和 a[4]。

（5）不能在方括号中用变量来表示元素的个数，但是可以是符号常数或常量表达式。例如：

```
#define CONST 5
void main()
{
    int a[13+10],b[12+ CONST];
        ……
}
```

是合法的。但是下述说明方式是错误的。

```
void main()
{
    int i=10;
    int a[i];
    ……
}
```

（6）允许在同一个类型说明中定义多个数组和变量。例如：

```
int a,b,k[10],m[20];
```

数组元素是组成数组的基本单元。数组元素也是一种变量，数组元素的一般形式为：

数组名[下标]

其中的下标表示元素在数组中的顺序号，只能为整型常量或整型表达式，如为小数时，C 编译将自动取整。

例如，a[10]、b[i+j]、c[i++]都是合法的数组元素。

数组元素通常也称为下标变量。必须先定义数组，才能使用下标变量。在 C 语言中只能逐个地使用下标变量，而不能一次引用整个数组。例如，输出有 10 个元素的数组必须

使用循环语句逐个输出各下标变量：

```
for(i=0; i<10; i++)
    printf("%d",a[i]);
```

而不能用一个语句输出整个数组。下面的写法是错误的：

```
printf("%d",a);
```

数组的赋值方法有以下两种：

①在程序执行过程中，用赋值语句对数组元素逐个赋值。

例如：

```
for(i=0; i<10; i++)
    a[i]=i;
```

②初始化赋值。

数组初始化赋值是指在数组说明时给数组元素赋予初值。数组初始化是在编译阶段进行的，这样将减少运行时间，提高效率。

初始化赋值的一般形式为：

```
static 类型说明符 数组名[常量表达式]={值，值……值};
```

其中 static 表示静态存储类型，C 语言规定只有静态存储数组和外部存储数组才可进行初始化赋值。在花括号中的各数据值即各元素的初值，各值之间用逗号间隔。例如：

```
static int a[10]={ 0,1,2,3,4,5,6,7,8,9 }; //相当于 a[0]=0;a[1]=1……a[9]=9;
```

C 语言对数组的初始赋值还有以下几点规定：

- 可以只给部分元素赋初值。当花括号中值的个数少于元素个数时，只给前面部分的元素赋值。例如：

```
static int a[10]={0,1,2,3,4};   //给 a[0]～a[4]5 个元素赋值,后 5 个元素自动赋 0。
```

- 只能给元素逐个赋值，不能给数组整体赋值。例如给 10 个元素全部赋 1 值，只能写为：static int a[10]={1,1,1,1,1,1,1,1,1,1};
  而不能写为：static int a[10]=1;
- 如不给可初始化的数组赋初值，则全部元素均为 0。
- 如给全部元素赋值，则在数组说明中，可以不给出数组元素的个数。例如：

```
static int a[5]={1,2,3,4,5};
```

可写为：

```
static int a[]={1,2,3,4,5};
```

**2. 二维数组**

数组可以是多维的，下面介绍二维数组。

二维数组说明的一般形式为：

类型说明符 数组名 [常量表达式][常量表达式]；

例如：

```
int a[3][6] ;        // 定义了 3 行 6 列共 18 个元素的二维数组 a[ ][ ]。
```

二维数组的存取顺序是：按行存取，先存取第 1 行元素的第 0 列，第 1 列，第 2 列……直到第 1 行的最后一列。然后返回到第 2 行开始，再存取第 2 行的第 0 列，第 1 列，第 2 列……直到第 2 行的最后一列。如此下去，直到最后一行的最后一列。

C 语言允许使用多维数组，理解了二维数组，进一步掌握多维数组并不困难。例如：

```
int a[2][4][6];          // 定义了一个类型为整型的三维数组。
```

以下是对二维数组的全部元素赋初值的两种方法：

①分行赋初值，例如：

```
int a[3][4]={{1,1,1,1},{2,2,2,2},{3,3,3,3}};
```

②用一个花括号，按数组排列顺序赋初值，例如：

```
int a[3][4]={1,1,1,1,2,2,2,2,3,3,3,3};
```

下面是二维数组部分赋初值的两个例子。

```
int a[3][4]={{1},{2},{3}};
```

其赋值后的数组元素如下：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \end{bmatrix}$$

```
int a[3][4]={{1},{},{3,4}};
```

其赋值后的数组元素如下：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 \end{bmatrix}$$

**3. 字符数组**

基本类型为字符类型的数组称为字符数组。显然字符数组是用来存放字符的。在字符数组中，一个元素存放一个字符，所以可以用字符数组来存储长度不同的字符串。

字符数组的定义与前文所述的数组定义方法类似。

例如：

```
char a[8];  // 定义一个有 8 个字符的一维字符数组。
```

字符数组赋初值的最直接的方法是将各字符逐个赋给数组中的各个元素。例如：

```
char a[8]={ `H`,`E`,`L`,`L`,`O`,`\0`};
```

定义了一个字符型数组 a[ ]，有 8 个数组元素，并且将 6 个字符（其中包括一个字符串结束标志`\0`）分别赋给了 a[0]~a[5]，剩余的 a[6]和 a[7]被系统自动赋予空格字符。

C 语言还允许用字符串直接给字符数组赋初值，有以下两种形式：

```
char a[8]={“HELLO”};
char a[8]= “HELLO”;
```

用双引号括起来的一串字符，称为字符串常量。C 编译器会自动地在字符串末尾加上结束符`\0`（NULL）。

**注意：**

用单引号括起来的字符为字符的 ASCII 码值，而不是字符串。比如`a`表示 a 的 ASCII 码值 97，而“a”表示一个字符串，它由两个字符 a 和\0 组成。

一个字符串可以用一维数组来装入，但数组的元素数目一定要比字符数多一个，以便 C 编译器自动在其后加入结束符`\0`。

若干个字符串可以装入一个二维字符数组中。数组的第一个下标是字符串的个数，第二个下标定义每个字符串的长度，该长度应当比这批字符串中最长的串多一个字符，用于装入字符串的结束符`\0`。比如 char a[20][61]定义了一个二维字符数组 a，它可容纳 20 个字符串，每串最长可达 60 个字符。

二维字符数组的第一个下标可以不给出，但必须给出第二个下标。

例如：

```
Uchar code msg[ ][20]=
{{“message 1”,\n},
 {“message 2”,\n},
 {“message 3”,\n}};
```

第一个下标可取默认值，它由数据常量表决定（此处为 3）；第二个下标不能从数据表中得到，因此必须给出。

#### 3.1.1.4 指针

指针是 C 语言中的一个重要概念，使用指针可以使得程序紧凑、高效。C 语言在处理指针时非常灵活，可以说掌握 C 语言的精华在于掌握指针。

**1. 基本概念**

指针是一个特殊的变量，它里面存储的数值被解释成为内存里的一个地址。指针定义的一般形式为：

```
类型识别符 * 指针变量名;
```

例如：

```
int * pointer;
```

**注意：**

指针变量名前的星号“*”表示该变量为指针变量，但指针变量名应该是 pointer，而不是* pointer。

要搞清一个指针需要知道四方面的内容：指针的类型、指针所指向的类型、指针的值或者指针所指向的内存区、指针本身所占据的内存区。

（1）指针的类型

从语法的角度考虑，只要把指针声明语句里的指针名字去掉，剩下的部分就是此指针的类型。这是指针本身所具有的类型。

例如：

```
int * ptr;          // 指针的类型是 int *
char * ptr;         // 指针的类型是 char *
int ** ptr;         // 指针的类型是 int **
int (* ptr)[3];     // 指针的类型是 int (*)[3]
int *(* ptr)[4];    // 指针的类型是 int *(*)[4]
```

（2）指针所指向的类型

当通过指针来访问指针所指向的内存区时，指针所指向的类型决定了编译器将把那片内存区里的内容当做什么类型来看待。从语法的角度考虑，只要把指针声明语句中的指针名字和名字左边的指针声明符“*”去掉，剩下的就是指针所指向的类型。

例如：

```
int * ptr;          // 指针所指向的类型是 int
char * ptr;         // 指针所指向的类型是 char
int ** ptr;         // 指针所指向的类型是 int *
int (* ptr)[3];     // 指针所指向的类型是 int ()[3]
int *(* ptr)[4];    // 指针所指向的类型是 int *()[4]
```

在指针的算术运算中，指针所指向的类型有很大的作用。指针的类型（即指针本身的类型）和指针所指向的类型是两个概念。把与指针搅和在一起的“类型”这个概念分成“指针的类型”和“指针所指向的类型”两个概念，是掌握指针的关键点之一。

（3）指针的值（或者称做指针所指向的内存区或地址）

指针的值是指针本身存储的数值，这个值将被编译器当做一个地址，而不是一个一般的数值。

指针所指向的内存区就是从指针的值所代表的那个内存地址开始的、长度为 sizeof（指针所指向的类型）的一片内存区。如果说一个指针的值是 X，那么相当于说该指针指向了以 X 为首地址的一片内存区域；如果说一个指针指向了某块内存区域，那么相当于说该指

针的值是这块内存区域的首地址。

指针所指向的内存区和指针所指向的类型是两个完全不同的概念。以 int *ptr 为例，指针所指向的类型已经有了，但由于指针还未初始化，所以它所指向的内存区是不存在的，或者说是无意义的。

每遇到一个指针，都应该问三个问题：这个指针的类型是什么？指针指向的类型是什么？该指针指向了哪里？

（4）指针本身所占据的内存区

指针本身占用了多大的内存只需用函数 sizeof（指针的类型）测一下就知道了。对于 8 位平台，指针本身占据了 1 个字节的长度；对于 16 位平台，指针本身占据了 2 个字节的长度；对于 32 位平台，指针本身占据了 4 个字节的长度。

在理解指针的概念时，务必弄清楚“变量的指针”和“指针变量”两个概念。变量的指针就是变量的地址。指针变量指的是指向变量的指针变量，若有一个变量专门用来存放另一个变量的地址（即指针），则该变量称为指向变量的指针变量（简称指针变量），指针变量的值是指针。

指针变量的引用是通过取地址运算符“&”来实现的。使用“&”和赋值运算符“＝”就可以使一个指针变量指向一个变量。例如：

```
int * ptr;
int a,b;
ptr = &a;
```

上述操作后，指针变量“ptr”就指向了变量“a”，即指针变量“ptr”所对应的内存地址单元中装入了变量“a”所对应的内存单元的地址。于是将整型变量“a”的值赋给整型变量“b”有了另一种方法：

```
b = * ptr;
```

**注意：**

星号“*”在指针变量定义时和在指针运算时所代表的含义是不同的。在进行指针变量定义时* ptr 中的星号“*”是指针变量类型说明符；在进行指针运算时 b = * ptr 中的星号“*”是指针运算符。

如果已经完成了指针变量的定义和引用，如：

```
int * ptr;
int a;
ptr = &a;
```

则在进行指针运算时有以下内涵：

- * ptr 与 a 是等价的，即* ptr 就是 a。
- 由于* ptr 与 a 等价，因此&* ptr 与&a 等价。
- 由于 ptr = &a ，则*& a 与*ptr 等价，即*& a 与 a 等价。

- *ap++相当于 a++。

**2. 数组指针和指向数组的指针变量**

数组指针就是数组的起始地址。若有一个变量，用做一个数组的起始地址（指针），则称它为指向数组的指针变量。

首先定义一个数组 a[10]和一个指向数组的指针变量 arrptr：

```
int a[10];              // a 为包含 10 个元素的数组
int * arrptr;           // arrptr 为指向整型数据的指针
```

为了将指针变量指向数组，需要对 arrptr 进行引用。有两种引用方法：

```
arrptr = &a[0];
```

或者

```
arrptr = a;          // 数组名 a 可代替数组 a[ ]的首地址，即第一个元素的地址
```

此时数组 a[10]的第一个元素 a[0]的地址就赋给了指针变量 arrptr，也就是指针变量 arrptr 指向数组 a[ ]的第 0 号元素 a[0]。

也可以将指针变量的定义和引用放在一个语句中，例如：

```
int * arrptr = &a[0];
```

或者

```
int * arrptr = a;
```

引用数组元素可以使用数组下标法，也可以使用指针法。指针法引用数组元素能使目标程序代码效率高，占用内存少，运行速度快。

通过指针引用数组的方法如下：

①arrptr+i 和 a+i 就是数组元素 a[i]的地址，即它们指向数组 a[ ]的第 *i* 个元素。由于 a 代表数组的首地址，则 a+i 与 arrptr+i 等价。

②*(arrptr+i)或*(a+i)是“arrpt+i”或“a+i”所指向的数组元素，即 a[i]。

③指向数组的指针变量可以带下标，如 arrptr[i]和*(arrptr+i)等价。

**注意：**

“arrptr+1”指向数组首地址的下一个元素，而不是将指针变量 arrptr 的值简单加 1。例如，若数组的类型是浮点型，每个数组元素占 4 个字节，则对于浮点型指针变量 arrptr 而言，“arrptr+1”意味着使 arrptr 的原值（地址）加 4 个字节，以使它指向下一元素。一般而言，“arrptr+i”所代表的地址实际上是“app_i×*k*”，*k* 是一个数组元素在内存中所占字节数（由元素类型来确定，如对于整型 *k*=2，对于实型 *k*=4，对字符型 *k*=1）。

指向数组的指针变量有以下一些运算：

①arrptr++（或 arrptr+=1）。将使指针变量 arrptr 指向下一个数组元素，即 a[1]。

②*arrptr++。由于++与*运算符优先级相同，而结合方向为自右向左，因此* arrptr ++等价于*( arrptr ++)，其作用是先得到 arrptr 指向的变量的值（即* arrptr），然后再执行 arrptr 自加运算。

③*++arrptr。*++arrptr 与*arrptr++不同，前者是先使 arrptr 自加 1 再取* arrptr 值，后者是先取* arrptr 值再使 arrptr 自加 1。

④(*arrptr)++。表示 arrptr 所指向的元素值加 1。注意不是指针变量的值加 1。

⑤当 arrptr 当前指向数组中第 *i* 个元素时：

*(arrptr--)与 a[i--]等价，相当于先执行* arrptr，然后再使 arrptr 自减；

*(++arrptr)与 a[++i]等价，相当于先执行 arrptr 自加，再执行* arrptr 运算；

*(--arrptr)与 a[--i]等价，相当于先执行 arrptr 自减，再执行* arrptr 运算。

#### 3.1.1.5 结构

将多个不同类型的变量（结构成员）结合在一起形成的一个组合型变量，称为结构变量，简称结构。结构成员可以是基本类型、枚举类型、指针类型、数组类型或其他结构类型的变量。结构元素的定义规则和变量命名相同。与数组不同，结构中的结构成员可以使用不同的数据类型。

**1. 结构类型及变量的定义**

定义结构类型的一般格式为：

```
struct 结构名
{
    结构成员说明
};
```

结构成员说明的格式为：

```
类型标识符  成员名;
```

**注意：**

结构名是结构的标识符，不是变量名。

结构成员和数组中的成员一样，但是数组中成员是以下标来访问的，而结构是按变量名字来访问成员的。

下面举一个例子来说明怎样定义结构类型：

```
struct employee
{
    char name[8];
    int age;
    char sex[2];
    char depart[20];
    float wage1, wage2, wage3, wage4, wage5;
} ;
```

这个例子定义了一个结构类型，名为 employee，struct 为结构类型定义的关键字，employee 为结构名。struct employee 是程序员自己定义的标准类型（和 int、char、float 等类型一样），可以用来定义变量的类型。

上面定义的 struct employee 只是结构体的类型名，而不是变量名。程序中还需进一步定义该结构类型的变量名。定义一个结构变量有以下 3 种方法：

（1）先定义结构的类型，再定义该结构的变量名

例如：

```
struct employee
{
    char name[8];
    int age;
    char sex[2];
    char depart[20];
    float wage1, wage2, wage3, wage4, wage5;
} ;
struct employee worker1,worker2;        // 定义结构的变量名
```

（2）在定义结构类型的同时定义该结构变量

例如：

```
struct employee
{
    char name[8];
    int age;
    char sex[2];
    char depart[20];
    float wage1, wage2, wage3, wage4, wage5;
} worker1,worker2;                    // 定义结构的变量名
```

（3）直接定义结构类型变量

例如：

```
struct                              // 此处省略结构名
{
  char name[8];
  int age;
  char sex[2];
  char depart[20];
  float wage1, wage2, wage3, wage4, wage5;
} worker1,worker2;                  // 定义结构的变量名
```

这种省略结构名的结构称为无名结构，这种情况常出现在函数内部。

**2. 结构类型变量的使用**

结构类型与结构类型变量是两个不同的概念，其可操作的对象是结构类型变量，而不

是结构类型。

结构不能作为一个整体参加赋值、存取和运算，也不能整体地作为函数的参数或函数的返回值。对结构所执行的操作只能用&运算符取结构的地址，或对结构变量的成员分别加以引用。引用的方式为：

结构变量名.成员名;

例如：

```
worker1.age =26;
```

其中的句点“.”是成员运算符，它在所有运算符中优先级最高，worker1.age 可作为一个变量来操作。

**3. 结构数组和结构指针**

结构是一种新的数据类型，同样可以有结构数组和结构指针。

结构数组就是具有相同结构类型的变量集合。假如要定义一个班级 30 个同学的姓名、性别、年龄和住址，可以定义成一个结构数组。如下所示：

```
struct
{
    char name[8];
    char sex[2];
    int age;
    char addr[40];
}student[30];
```

也可定义为：

```
struct string
{
    char name[8];
    char sex[2];
    int age;
    char addr[40];
};
struct string student[30];
```

需要指出的是结构数组成员的访问是以数组元素为结构变量的，其形式为：

结构数组元素.成员名

例如：

```
student[0].name
student[20].age
```

实际上结构数组相当于一个二维构造，第一维是结构数组元素，每个元素是一个结构变量，第二维是结构成员。

**注意：**

结构数组的成员也可以是数组变量。

例如：

```
struct a
{
    int m[3][5];
    float f;
    char s[20];
}y[4];
```

为了访问结构 a 中结构变量 y[2]的这个变量，可写成：

```
y[2].m[1][4]
```

结构指针是指向结构的指针。它由一个加在结构变量名前的星号“*”操作符来定义，例如用前面已说明的结构定义一个结构指针如下：

```
struct string
{
    char name[8];
    char sex[2];
    int age;
    char addr[40];
}*student;
```

也可省略结构指针名只进行结构说明，然后再用下面的语句定义结构指针：

```
struct string *student;
```

使用结构指针对结构成员的访问，与结构变量对结构成员的访问在表达方式上有所不同。结构指针对结构成员的访问表示为：

结构指针名->结构成员

例如要给上面定义的结构中的 name 和 age 赋值，可以用下面的语句：

```
strcpy(student->name, "Zhang TF");
student->age=18;
```

实际上，student->name 就是(*student).name 的缩写形式。

需要指出的是结构指针是指向结构的一个指针，即结构中第一个成员的首地址，因此在使用之前应该对结构指针初始化，即分配整个结构长度的字节空间，这可用下面的函数完成，仍以上例来说明如下：

```
student=(struct string*)malloc(size of (struct string));
```

size of (struct string)自动求取 string 结构的字节长度，malloc()函数定义了一个大小为结构长度的内存区域，然后将其首地址作为结构指针返回。

需要注意以下两点：

- 结构作为一种数据类型，定义的结构变量或结构指针变量同样有局部变量和全程变量之分，视定义的位置而定。
- 结构变量名不是指向该结构的地址，这与数组名的含义不同，因此若需要求结构中第一个成员的首地址应该使用语句：&[结构变量名]。

#### 3.1.1.6 共用体

在 C 语言中，不同数据类型的数据在编译时占据各自拥有的内存空间，彼此不能共享空间。但共用体类型（也称联合类型）是个例外。共用体是 C 语言构造类型数据结构之一，C 编译器在编译时为此类型指定一块内存空间，并允许各种类型的数据共同使用。

共用体与结构类似，它也可以包含多个不同数据类型的元素，但其变量所占有的内存空间并不是各成员所需存储空间的总和，而是在任何时候其变量至多只能存放该类型所包含的一个成员，即它所包含的各个成员只能分时共享同一存储空间。这是共用体和结构的区别所在。共用体类型是指将不同的数据项组织成一个整体，它们在内存中占用同一段存储单元。

共用体类型的定义形式如下：

```
union 共用体类型标识符
{
    类型说明符  变量名;
};
```

例如：

```
union data
{
    int a;
    float b;
    char c;
}
```

共用体类型 data 包含 3 个不同类型的元素，分别是 int、float 和 char 类型。

定义共用体变量的一般形式为：

```
union 共用体类型标识符 共用体变量名表;
```

例如：

```
union data obj;
```

obj 为 data 类型共用体变量，它能使一个整型变量 obj.a、一个实型变量 obj.b 和一个字符型变量 obj.c 分时共享同一存储空间。

与结构变量一样，也可在定义共用体类型的同时定义共用体变量。

例如：

```
union data
{
    int a;
    float b;
    char c;
} obj;                    // 定义共用体变量
```

或者

```
union                     // 此处无共用体类型标识符
{
    int a;
    float b;
    char c;
} obj;                    // 定义共用体变量
```

对于共用体变量，系统只给该变量按其各共用体成员中所需空间最大的那个成员的长度分配内存空间。

对共用体的成员的引用与结构体成员的引用相同。但由于共用体各成员共用同一段内存空间，使用时需根据需要使用其中的某一个成员。共用体的这一特点，方便程序设计人员在同一内存区对不同数据类型的交替使用，增加灵活性、节省内存。

可以引用共用体变量的成员，其用法与结构体完全相同。若定义共用体类型为：

```
union data
{
    int a;
    float b;
    char c;
} obj;
```

其成员引用为：obj.a、obj.b、obj.c。注意，不能同时引用 3 个成员，在某一时刻，只能使用成员之一。

共用体变量可以在不同时间内保存不同类型和长度的数据，从而提供了在同一存储单元中可以分时操作不同类型数据的功能。

#### 3.1.1.7　枚举

在实际问题中，有些变量的取值被限定在一个有限的范围内。例如，发光二极管有红、绿两色，一个星期有 7 天，一年有 12 个月，一个班每周有 6 门课程等。如果把这些量说明为整型、字符型或其他类型显然是不妥当的。为此，C 语言提供了一种称为“枚举”的类型。在枚举类型的定义中列举出所有可能的取值，被说明为该枚举类型的变量取值不能超过定义的范围。

枚举类型定义的一般形式为：

```
enum 枚举名
    { 枚举值列表 };
```

在枚举值列表中应罗列出所有可用值，这些值也称为枚举元素。

例如：

```
enum weekday
    { sun,mou,tue,wed,thu,fri,sat };
```

该枚举名为 weekday，枚举值共有 7 个，即一周中的 7 天。凡被说明为 weekday 类型变量的取值只能是 7 天中的某一天。

枚举变量的定义如同结构和共用体一样，也可用不同的方式说明：先定义后说明、同时定义说明或直接说明。

例如：

```
enum weekday
    { sun,mou,tue,wed,thu,fri,sat };
enum weekday a,b,c;
```

或者：

```
enum weekday
    { sun,mou,tue,wed,thu,fri,sat } a,b,c;
```

或者：

```
enum
    { sun,mou,tue,wed,thu,fri,sat } a,b,c;
```

枚举类型在使用中有以下规定：

①枚举值是常量，不是变量。不能在程序中用赋值语句再对它赋值。

②枚举元素本身由系统定义了一个表示序号的数值，从 0 开始顺序定义为 0，1，2……例如在 weekday 中，sun 值为 0，mon 值为 1……sat 值为 6。

③只能把枚举值赋予枚举变量，不能把元素的数值直接赋予枚举变量。

例如：a=sum;b=mon; 是正确的，而 a=0;b=1; 是错误的。

如果一定要把数值赋予枚举变量，必须用强制类型转换，例如：

```
a=(enum weekday)2;  //将顺序号为 2 的枚举元素赋予枚举变量 a
```

上句语句相当于：

```
a=tue;
```

④枚举元素不是字符常量也不是字符串常量，使用时不要加单、双引号。

### 3.1.2 运算符与表达式

C 语言中的运算符和表达式数量之多，在高级语言中是少见的。正是丰富的运算符和

表达式使 C 语言的功能十分完善。这也是 C 语言的主要特点之一。

#### 3.1.2.1　运算符分类与优先级

C 语言的运算符主要可分为以下 7 类：

（1）算术运算符

用于各类数值运算，详见下文。

（2）关系运算符

用于比较运算，详见下文。

（3）逻辑运算符

用于逻辑运算，详见下文。

（4）位操作运算符

参与运算的量按二进制位进行运算，详见下文。

（5）赋值运算符

用于赋值运算，详见下文。

（6）条件运算符

唯一的三目运算是条件运算，条件运算符是“?:”。条件表达式的形式为：

<表达式 1>？<表达式 2>：<表达式 3>

其含义为：若<表达式 1>的值为“真”，则条件表达式取<表达式 2>的值；否则取<表达式 3>的值。

（7）指针运算符

用于取内容（*）和取地址（&）两种运算。这在前面介绍指针类型时已经详细介绍了。

C 语言中，运算符的运算优先级共分为 15 级。1 级为最高级，15 级为最低级。在表达式中，优先级较高的先于优先级较低的进行运算。而在一个运算量两侧的运算符优先级相同时，则按运算符的结合性所规定的结合方向处理。C 语言中各运算符的结合性分为两种，即左结合性（自左至右）和右结合性（自右至左）。例如算术运算符的结合性是自左至右，即先左后右。如有表达式“x-y+z”，则“y”应先与减号“-”结合，执行“x-y”运算，然后再执行“+z”的运算。这种自左至右的结合方向就称为“左结合性”。而自右至左的结合方向称为“右结合性”。最典型的右结合性运算符是赋值运算符。如“x=y=z”，由于“=”的右结合性，应先执行“y=z”再执行“x=(y=z)”运算。C 语言运算符中有不少为右结合性，应注意区别，以避免理解错误。

#### 3.1.2.2　算术运算符与表达式

算术运算符用于各类数值运算。包括加（+）、减（-）、乘（*）、除（/）、求余（或称模运算，%）、自增（++）、自减（--）共 7 种。用算术运算符和括号将运算对象连接起来的式子称为算术表达式。其中运算对象包括常量、变量、函数、数组、结构等。

加法运算符“+”为双目运算符，即应有两个量参与加法运算。如 a+b，4+8 等。具有左结合性。

减法运算符“-”为双目运算符。但减号“-”也可作为负值运算符，此时为单目运算，如-x，-5 等，具有左结合性。

乘法运算符“*”为双目运算符，具有左结合性。

除法运算符“/”为双目运算符，具有左结合性。参与运算量均为整型时，结果也为整型，舍去小数。如果运算量中有一个是实型，则结果为双精度实型。

求余运算符（模运算符）“%”为双目运算符，具有左结合性。要求参与运算的量均为整型。求余运算的结果等于两数相除后的余数。

自增 1 运算符记为“++”，其功能是使变量的值自增 1。自减 1 运算符记为“--”，其功能是使变量值自减 1。自增 1 和自减 1 运算符均为单目运算，都具有右结合性。可有以下几种形式：

++i：i 自增 1 后再参与其他运算；

--i：i 自减 1 后再参与其他运算；

i++：i 参与运算后，i 的值再自增 1；

i--：i 参与运算后，i 的值再自减 1。

在理解和使用上容易出错的是 i++和 i--，特别是当它们出现在较复杂的表达式或语句中时，常常难以弄清，因此应仔细分析。

#### 3.1.2.3 关系运算符与表达式

关系运算符用于比较运算。包括大于（>）、小于（<）、大于等于（>=）、小于等于（<=）、等于（==）和不等于（!=）6 种。

前 4 种优先级相同，后 2 种优先级相同，前 4 种的优先级又高于后 2 种。关系运算符的优先级低于算术运算符，但高于赋值运算符。

例如：

c>a+b　　等效于　c>(a+b)

a>b!=c　　等效于　(a>b)!=c

关系运算符的结合性为左结合。

用关系运算符将两个表达式（算术表达式、关系表达式、逻辑表达式等）连接起来的式子，称为关系表达式。关系表达式的结果为逻辑真或假。C 语言以 1 代表真，0 代表假。

例如，若 a=5，b=3，c=0，则：

a>b 的值为真，表达式的值为 1；

d=a>b，d 的值为 1；

b+c<a 的值为假，表达式的值为 0；

e=a>b>c，由于结合性为左结合，故 a>b 的值为 1，而 1>c 的值为 0，因此 e 的值为 0。

#### 3.1.2.4　逻辑运算符与表达式

逻辑运算符用于逻辑运算。包括逻辑与（&&）、逻辑或（||）和逻辑非（!）3 种。

符号“&&”和“||”是双目运算符，要求有两个运算对象，而惊叹号“!”为单目运算符，只要求有一个运算对象。“!”运算符优先级高于算术运算符，算术运算符优先级高于关系运算符，关系运算符优先级高于逻辑运算符“&&”和“||”，“&&”和“||”的优先级高于赋值运算符。

用逻辑运算符将关系表达式或逻辑量连接起来的式子称为逻辑表达式。逻辑表达式的结合性为自左向右，其值应该是一个逻辑的真或假。逻辑表达式的值和关系表达式的值相同，以 1 代表真，以 0 代表假。

若 a=8，b=6，则：

因为 a=8 为真，所以!a 为假（0）；

因为 a、b 均为真，所以 a||b 为真（1），a&&b 为真（1）；

因为“!”的优先级高于“&&”，所以!a&&b 为假（0）。

#### 3.1.2.5　位操作运算符与表达式

位操作运算符是用来进行二进制位运算的运算符，包括逻辑位运算符和移位运算符。逻辑位运算符是位与（&）、位或（|）、位取反（~）和位异或（^）；移位运算符是位左移（<<）和位右移（>>）。

除了位取反（~）是单目运算符外，其他位操作运算符均为双目运算符。

位取反（~）用来将二进制数按位取反，即 1 变 0，0 变 1。位取反（~）运算符的优先级比别的算术运算符、关系运算符和其他运算符都高。

位与运算符（&）的运算规则如下：参与运算的两个运算对象，若两者相应的位都为 1，则该位结果值为 1，否则为 0。

位或运算符（|）的运算规则如下：参与运算的两个运算对象，若两者相应的位都为 0，则该位结果值为 0，否则为 1。

位异或运算符（^）的运算规则如下：参与运算的两个运算对象，若两者相应的位值相同，则结果值为 0；若两者相应的位值不同，则结果值为 1。

位左移运算符（<<）、位右移运算符（>>）用来将一个数的二进制位全部左移或右移若干位，移位后，空白位补 0，而溢出的位舍弃。

例如：若 a=ABH=10101011B，则：

a=a<<2，将 a 值左移 2 位，其结果为 10101100B=ACH；

a=a>>2，将 a 值右移 2 位，其结果为 00101010B=2AH。

#### 3.1.2.6　赋值运算符与表达式

赋值运算符用于赋值运算，分为简单赋值（=）、复合算术赋值（+=，-=，*=，/=，%=）

和复合位运算赋值（&=，|=，^=，>>=，<<=）3 类共 11 种。

简单赋值运算符记为“=”。由“=”连接的式子称为赋值表达式。其一般形式为：

变量=表达式

例如：

```
x=a+b
w=sin(a)+sin(b)
```

赋值表达式的功能是计算表达式的值再赋予左边的变量。赋值运算符具有右结合性。因此“a=b=c=5”可理解为“a=(b=(c=5))”。

在其他高级语言中，赋值构成了一个语句，称为赋值语句。而在 C 语言中，把等于符号“=”定义为运算符，从而组成赋值表达式。凡是表达式可以出现的地方均可出现赋值表达式。例如，式子“x=(a=3)+(b=6)”是合法的。它的意义是把 3 赋予“a”，6 赋予“b”，再把“a”和“b”相加，其和赋予“x”，故“x”应等于 9。

如果赋值运算符两边的数据类型不相同，则系统将自动进行类型转换，即把赋值号右边的类型换成左边的类型。具体规定如下：

①实型赋予整型，舍去小数部分。

②整型赋予实型，数值不变，但将以浮点形式存放，即增加小数部分（小数部分的值为 0）。

③字符型赋予整型，由于字符型为一个字节，而整型为两个字节，故将字符的 ASCII 码值放到整型量的低 8 位中，高 8 位为 0。

④整型赋予字符型，只把低 8 位赋予字符量。

在赋值符“=”之前加上其他二目运算符可构成复合赋值符。如+=、-=、*=、／=、%=、<<=、>>=、&=、^=和|=。构成复合赋值表达式的一般形式为：

变量 双目运算符=表达式

它等效于：

变量=变量 运算符 表达式

例如：

“a+=5”等价于“a=a+5”；

“x*=y+7”等价于“x=x*(y+7)”；

“r%=p”等价于“r=r%p”。

复合赋值符这种写法，对初学者可能不习惯，但十分有利于编译处理，能提高编译效率并产生质量较高的目标代码。

### 3.1.3 程序结构与函数

单片机的 C 语言程序结构与一般 C 语言没有什么差别，也是函数定义的集合体。集合

中仅有一个名为 main 的主函数。主函数是程序的入口，其中的所有语句执行完毕，则程序执行结束。

#### 3.1.3.1　程序结构

C 语言程序的一般组成结构如下：

```
全局变量声明
main( )                          // 主函数
{
    局部变量声明
    执行语句
}

函数 1（形参表）                  // 函数 1
形参声明
{
    局部变量声明
    执行语句
}
……

函数 n（形参表）                  // 函数 n
形参声明
{
    局部变量声明
    执行语句
}
```

C 程序的执行从 main( )函数开始，调用其他函数后返回到主函数 main( )中，最后在 main( )函数中结束整个程序的运行。

C 语言程序是由函数组成的，函数是 C 程序的基本构成单位，下面来介绍函数。

#### 3.1.3.2　函数

所有的函数在定义时都是相互独立的，函数中不能定义其他函数，即不能嵌套定义，但函数可以互相调用。习惯上把调用者称为主调函数。函数还可以自己调用自己，称为递归调用。函数调用的一般规则是：主函数可以调用其他普通函数，普通函数间可以互相调用，但普通函数不能调用主函数。

**1. 函数分类**

可以从不同角度对函数进行分类。

①从函数定义的角度看，函数可分为库函数和用户定义函数两种。

- 标准库函数

它由 C 编译系统的函数库提供，用户无需定义，也不必在程序中作类型说明，只需在

程序前包含有该函数原型的头文件即可在程序中直接调用。

- 用户自定义函数

它是由用户按需要编写的函数。对于用户自定义函数，不仅要在程序中定义函数本身，而且在主调函数模块中还必须对该被调函数进行类型说明，然后才能使用。

②C 语言函数兼有其他语言中的函数和过程两种功能，从这个角度看，又可把函数分为有返回值函数和无返回值函数两种。

- 有返回值函数

此类函数被调用执行完后将向调用者返回一个执行结果，称为函数返回值，如数学函数即属于此类函数。由用户定义的这种要返回函数值的函数，必须在函数定义和函数说明中明确返回值的类型。

- 无返回值函数

此类函数用于完成某项特定的处理任务，执行完成后不向调用者返回函数值。这类函数类似于其他语言的过程。由于函数不需要返回值，用户在定义此类函数时可指定它的返回为“空类型”，空类型的说明符为“void”。

③从主调函数和被调函数之间数据传送的角度看又可分为无参函数和有参函数两种。

- 无参函数

无参函数指函数定义、函数说明及函数调用中均不带参数的函数；主调函数和被调函数之间不进行参数传送。此类函数通常用来完成一组指定的功能，可以返回或不返回函数值。

- 有参函数

有参函数也称为带参函数。在函数定义及函数说明时都有参数，称为形式参数（简称为形参）。在函数调用时也必须给出参数，称为实际参数（简称为实参）。进行函数调用时，主调函数将把实参的值传送给形参，供被调函数使用。

**2. 函数定义**

函数由类型说明符、函数名、参数表和函数体四部分组合而成。函数名是一个标识符（大小写有区别，最长为 255 个字符）；参数表是用括号括起来的若干参数，项与项之间用逗号隔开；函数体是用大括号括起来的若干 C 语句，语句与语句之间用分号隔开；最后一个语句一般是 return（主函数中省略）；每个函数都返回一个值，该值由 return 语句中的表达式指定（省略时为零）；函数的类型就是返回值的类型，函数类型除整型外都需在函数名前加以指定。

（1）无参函数

无参函数的一般形式如下：

```
类型说明符 函数名( )
{
    类型说明
    语句
}
```

函数名是由用户定义的标识符，函数名后有一个空括号，其中无参数，但括号不可少。

大括号中的内容称为函数体。在函数体中也有类型说明，这是对函数体内部所用到的变量的类型说明。在很多情况下都不要求无参函数有返回值，此时函数类型符可以写为 void。

（2）有参函数

有参函数一般形式如下：

```
类型说明符 函数名(形参表)
形参类型说明
{
    类型说明
    语句
}
```

有参函数比无参函数多了两个内容，其一是形参表，其二是形参类型说明。在形参表中给出的参数称为形参（形式参数），它们可以是各种类型的变量，各参数之间用逗号间隔。在进行函数调用时，主调函数将赋予这些形式参数实际的值。形参既然是变量，当然必须给予类型说明。例如，定义一个函数，用于求两个数中的大数，可写为：

```
int max(a,b)
int a,b;
{
    if (a>b) return a;
    else return b;
}
```

第一行说明 max 函数是一个整型函数，其返回函数值是一个整数，形参为 a 和 b。第二行说明 a 和 b 均为整型量。a 和 b 的具体值是由主调函数在调用时传送过来的。在大括号“{}”中的函数体内，除形参外没有使用其他变量，因此只有语句而没有变量类型说明。上边这种定义方法称为“传统格式”。这种格式不易于编译系统检查，从而会引起一些非常细微而且难以跟踪的错误。ANSI C 的新标准中把对形参的类型说明合并到形参表中，称为“现代格式”。

例如 max 函数用现代格式可定义为：

```
int max(int a,int b)
{
    if(a>b) return a;
    else return b;
}
```

现代格式在函数定义和函数说明（后面将要介绍）时，给出了形参及其类型，在编译时易于对它们进行查错，从而保证了函数说明和定义的一致性。在 max 函数体中的 return 语句是把 a（或 b）的值作为函数的值返回给主调函数的。有返回值的函数中至少应有一个 return 语句。在 C 程序中，一个函数的定义可以放在任意位置，既可放在主函数 main 之前，也可放在 main 之后。

**3. 函数调用**

程序中是通过对函数的调用来执行函数体的，其过程与其他语言的子程序调用相似。C 语言中，函数调用的一般形式为：

函数名（实际参数表）

对无参函数调用时则无实际参数表。

实际参数表中的参数可以是常数、变量或其他构造类型数据及表达式。各实参之间用逗号分隔。在 C 语言中，可以用以下几种方式调用函数。

（1）函数表达式

函数作为表达式中的一项出现在表达式中，以函数返回值参与表达式的运算。这种方式要求函数是有返回值的。

例如：z=max(x,y)是一个赋值表达式，把 max 的返回值赋予变量 z。

（2）函数语句

函数调用的一般形式加上分号即构成函数语句。

例如：printf ("%D",a);scanf ("%d",&b);都是以函数语句的方式调用函数的。

（3）函数实参

函数作为另一个函数调用的实际参数出现。这种情况下把该函数的返回值作为实参进行传送，因此要求该函数必须是有返回值的。

例如：printf("%d",max(x,y)); 即把 max 调用的返回值又作为 printf 函数的实参来使用。

在函数调用中还应该注意的一个问题是求值顺序的问题。所谓求值顺序是指对实参表中各量是自左至右使用还是自右至左使用。各系统对此的规定不一定相同。

C 语言中规定在以下几种情况时可以省去主调函数中对被调函数的函数说明。

①如果被调函数的返回值是整型或字符型时，可以不对被调函数作说明而直接调用。这时系统将自动对被调函数返回值按整型处理。

②当被调函数的函数定义出现在主调函数之前时，在主调函数中也可以不对被调函数再作说明而直接调用。

③如果在所有函数定义之前，在函数外预先说明了各个函数的类型，则在以后的各主调函数中，可不再对被调函数作说明。例如：

```
char str(int a);
float func(float b);
main()
{
    ……
}
char str(int a)
{
    ……
}
```

```
float func(float b)
{
    ……
}
```

其中第一、第二行对 str 函数和 func 函数预先作了说明，因此在以后各函数中无需对 str 和 func 函数再作说明就可直接调用。

④对库函数的调用不需要再作说明，但必须把该函数的头文件用 include 命令包含在源文件前部。

**4. 函数参数与函数的值**

函数的参数分为形参和实参两种。形参出现在函数定义中，在整个函数体内都可以使用，离开该函数则不能使用；实参出现在主调函数中，进入被调函数后，实参变量也不能使用。形参和实参的功能是作为数据传送的。发生函数调用时，主调函数把实参的值传送给被调函数的形参，从而实现主调函数向被调函数的数据传送。

函数的形参和实参具有以下特点：

①形参变量只有在被调用时才分配内存单元，在调用结束时，即刻释放所分配的内存单元。因此，形参只在函数内部有效。函数调用结束返回主调函数后则不能再使用该形参变量。

②实参可以是常量、变量、表达式、函数等，无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值，以便把这些值传送给形参。因此应预先使用赋值、输入等办法使实参获得确定值。

③实参和形参在数量、类型和顺序上应严格一致，否则会发生“类型不匹配”的错误。

④函数调用中发生的数据传送是单向的。即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化。

下例可以说明这个问题：

```
void main()
{
    int n;
    printf("input number\n");
    scanf("%d",&n);
    s(n);
    printf("n=%d\n",n);
}
int s(int n)
{
    int i;
    for(i=n-1;i>=1;i--)
```

```
        n=n+i;
    printf("n=%d\n",n);
}
```

上述程序中定义了一个函数 s。在主函数中输入 n 值，并作为实参，在调用时传送给 s 函数的形参量 n（注意，本例的形参变量和实参变量的标识符都为 n，但这是两个不同的量，各自的作用域不同）。在主函数中用 printf 语句输出一次 n 值，这个 n 值是实参 n 的值。在函数 s 中也用 printf 语句输出了一次 n 值，这个 n 值是形参最后取得的 n 值（0）。从运行情况看，输入 n 值为 100。即实参 n 的值为 100。把此值传给函数 s 时，形参 n 的初值也为 100，在函数执行过程中，形参 n 的值变为 5050。返回主函数之后，输出实参 n 的值仍为 100。可见实参的值不随形参的变化而变化。

函数的值是指函数被调用之后，执行函数体中的程序段所取得的并返回给主调函数的值，如调用正弦函数取得正弦值。对函数的值（或称函数返回值）有以下一些说明：

- 函数的值只能通过 return 语句返回主调函数。在函数中允许有多个 return 语句，但每次调用只能有一个 return 语句被执行，因此只能返回一个函数值。
- 函数的值的类型和函数定义中函数的类型应保持一致。如果两者不一致，则以函数类型为准，自动进行类型转换。
- 如函数的值为整型，在函数定义时可以省去类型说明。
- 不返回函数值的函数，可以明确定义为“空类型”，类型说明符为 void。一旦函数被定义为空类型后，就不能在主调函数中使用被调函数的函数值了。为了使程序有良好的可读性并减少出错，凡不要求返回值的函数都应定义为空类型。

**5. 数组作为函数参数**

数组可以作为函数的参数使用，进行数据传送。数组用做函数参数有两种形式：一种是把数组元素（下标变量）作为实参使用；另一种是把数组名作为函数的形参和实参使用。

（1）数组元素作为函数实参

数组元素就是下标变量，它与普通变量并无区别。因此它作为函数实参使用与普通变量是完全相同的。在发生函数调用时，把作为实参的数组元素的值传送给形参，实现单向的值传送。

（2）数组名作为函数参数

用数组名作为函数参数与用数组元素作为实参有几点不同：

①用数组元素作为实参时，只要数组类型和函数的形参变量的类型一致，那么作为下标变量的数组元素的类型也和函数形参变量的类型是一致的，因此，并不要求函数的形参也是下标变量。换句话说，对数组元素的处理是按普通变量对待的。用数组名作为函数参数时，则要求形参和相对应的实参都必须是类型相同的数组，都必须有明确的数组说明。当形参和实参二者不一致时，会发生错误。

②当普通变量或下标变量作为函数参数时，形参变量和实参变量是由编译系统分配的

两个不同的内存单元。在函数调用时发生的值传送是把实参变量的值赋予形参变量。在用数组名作为函数参数时，不是进行值的传送，即不是把实参数组的每一个元素的值都赋予形参数组的各个元素，因为实际上形参数组并不存在，编译系统不为形参数组分配内存。那么，数据的传送是如何实现的呢？在前文中曾介绍过，数组名就是数组的首地址，因此在用数组名作为函数参数时所进行的传送只是地址的传送，也就是说把实参数组的首地址赋予形参数组名。形参数组名取得该首地址之后，也就等于有了实在的数组。实际上形参数组和实参数组为同一数组，共同拥有一段内存空间。

③前面已经介绍过，当变量作为函数参数时，所进行的值传送是单向的，即只能从实参传向形参，不能从形参传回实参。形参的初值和实参相同，而形参的值发生改变后，实参并不变化，两者的终值是不同的。而当用数组名作为函数参数时，情况则不同。由于实际上形参和实参为同一数组，因此当形参数组发生变化时，实参数组也随之变化。当然这种情况不能理解为发生了“双向”的值传递。但从实际情况来看，调用函数之后实参数组的值将由于形参数组值的变化而变化。

用数组名作为函数参数时还应注意以下几点：

- 形参数组和实参数组的类型必须一致，否则将引起错误。
- 形参数组和实参数组的长度可以不相同，因为在调用时，只传送首地址而不检查形参数组的长度。当形参数组的长度与实参数组不一致时，虽不致于出现语法错误（编译能通过），但程序执行结果将与实际不符，这是应予以注意的。
- 在函数形参表中，允许不给出形参数组的长度，或用一个变量来表示数组元素的个数。
- 多维数组也可以作为函数的参数。在函数定义时对形参数组可以指定每一维的长度，也可省去第一维的长度。

**6. 函数的嵌套与递归调用**

C 语言中不允许进行嵌套的函数定义，因此各函数之间是平行的，不存在上一级函数和下一级函数的问题。但是 C 语言允许在一个函数的定义中出现对另一个函数的调用，这样就出现了函数的嵌套调用，即在被调函数中又调用其他函数。这与其他语言的子程序嵌套的情形是类似的。

一个函数在它的函数体内调用它自身的情形称为递归调用。这种函数称为递归函数。C 语言允许函数的递归调用。在递归调用中，主调函数又是被调函数。执行递归函数将反复调用其自身，每调用一次就进入新的一层，例如有函数 func 如下：

```
int func (int x)
{
    int y;
    z=func(y);
    return z;
}
```

此函数是一个递归函数。但是运行该函数将无休止地调用其自身，这当然是不正确的。为了防止递归调用无终止地进行，必须在函数内有终止递归调用的手段。常用的办法是加条件判断，满足某种条件后就不再进行递归调用，然后逐层返回。以下用递归法计算 *n*! 为例说明递归调用的执行过程。

递归法计算 *n*!可用下述公式表示：

$n!=1 \quad (n=0,1)$

$n\times(n-1)! \quad (n>1)$

按公式可编程如下：

```
long func(int n)
{
    long f;
    if (n<0) printf("n<0,input error");
    else if (n==0||n==1) f=1;
    else f=func(n-1)*n;
    return(f);
}
main()
{
    int n;
    long y;
    printf("\ninput a inteager number:\n");
    scanf("%d",&n);
    y=func(n);
    printf("%d!=%ld",n,y);
}
```

程序中给出的函数 func 是一个递归函数。主函数调用 func 后即进入函数 func 执行，如果 n<0、n==0 或 n==1 时都将结束函数的执行，否则就递归调用函数自身。由于每次递归调用的实参为 n-1，即把 n-1 的值赋予形参 n，最后当 n-1 的值为 1 时再作递归调用，形参 n 的值也为 1，将使递归终止，然后可逐层退回。

### 3.1.4 流程控制语句

从程序流程的角度来看，程序可以分为 3 种基本结构：顺序结构、分支结构和循环结构。这 3 种基本结构可以组成所有的各种复杂程序。

顺序结构是一种最基本、最简单的编程结构。在这种结构中，程序由低地址向高地址顺序执行指令代码。

流程控制语句用于控制程序的流程，以实现程序的各种结构方式。

C 语言有 9 种流程控制语句，可分成以下 3 类：

①选择语句，如：if 语句、switch 语句。

②循环语句，如：do while 语句、while 语句、for 语句。

③转移语句，如：break 语句、goto 语句、continue 语句、return 语句。

#### 3.1.4.1　选择语句

选择语句使得单片机具有决策能力，它包括 if 语句和 switch 语句。

**1. if语句**

用 if 语句可以构成分支结构。它根据给定的条件进行判断，以决定执行某个分支程序段。C 语言的 if 语句有 3 种形式：基本形式、if-else 形式、if-else-if 形式。

（1）基本形式

其语法结构为：

```
if(表达式) 语句;
```

其语义是：如果表达式的值为真，则执行其后的语句，否则不执行该语句。例如：

```
void main()
{
    int a,b,max;
    printf("\n input two numbers: ");
    scanf("%d%d",&a,&b);
    max=a;
    if (max<b) max=b;
    printf("max=%d",max);
}
```

本例程序中，输入两个数 a 和 b，把 a 先赋给变量 max，再用 if 语句判别 max 和 b 的大小，如 max 小于 b，则把 b 赋予 max。因此 max 中总是大数，最后输出 max 的值。

（2）if-else 形式

其语法结构如下：

```
if (表达式)
    语句 1;
else
    语句 2;
```

其语义是：如果表达式的值为真，则执行“语句 1”，否则执行“语句 2”。例如：

```
void main()
{
    int a, b;
    printf("input two numbers: ");
    scanf("%d%d",&a,&b);
    if(a>b)
        printf("max=%d\n",a);
    else
```

```
        printf("max=%d\n",b);
    }
```

本例改用 if-else 语句判别 a 和 b 的大小，若 a 大，则输出 a，否则输出 b。

（3）if-else-if 形式

前两种形式的 if 语句一般都用于两个分支的情况。当有多个分支选择时，可采用 if-else-if 语句，其一般形式为：

```
if(表达式 1)
    语句 1;
else if(表达式 2)
    语句 2;
else if(表达式 3)
    语句 3;
......
else if(表达式 m)
    语句 m;
else
    语句 n;
```

其语义是：依次判断表达式的值，当出现某个值为真时，则执行其对应的语句，然后跳到整个 if 语句之外继续执行程序。如果所有的表达式均为假，则执行语句 n。然后继续执行后续程序。例如：

```
#include"stdio.h"
void main()
{
    char c;
    c=getchar();
    if(c<32)
        printf("This is a control character\n");
    else if(c>='0'&&c<='9')
        printf("This is a digit\n");
    else if(c>='A'&&c<='Z')
        printf("This is a capital letter\n");
    else if(c>='a'&&c<='z')
        printf("This is a small letter\n");
    else
        printf("This is an other character\n");
}
```

本例要求判别键盘输入字符的类别。可以根据输入字符的 ASCII 码来判别类型：由 ASCII 码表可知 ASCII 值小于 32 的为控制字符；在“0”和“9”之间的为数字；在“A”和“Z”之间的为大写字母；在“a”和“z”之间的为小写字母，其余则为其他字符。这是一个多分支选择的问题，用 if-else-if 语句编程可轻松实现。

当 if 语句中的执行语句又是 if 语句时，则构成了 if 语句嵌套的情形，采用嵌套结构实质上是为了进行多分支选择。在嵌套内的 if 语句可能又是 if-else 型的，这将会出现多个 if 和多个 else 重叠的情况，这时要特别注意 if 和 else 的配对问题。C 语言规定，else 总是与它前面最近的 if 配对。在一般情况下较少使用 if 语句的嵌套结构，因为这样会使得程序不便于阅读。

**2. switch语句**

C 语言还提供了另一种用于多分支选择的 switch 语句，其一般形式为：

```
switch(表达式)
{
    case 常量表达式 1: 语句 1;
    case 常量表达式 2: 语句 2;
    …
    case 常量表达式 n: 语句 n;
    default : 语句 n+1;
}
```

其语义是：计算表达式的值，并逐个与其后的常量表达式值相比较；当表达式的值与某个常量表达式的值相等时，即执行其后的语句，然后不再进行判断，继续执行后面所有 case 后的语句；如表达式的值与所有 case 后的常量表达式均不相同时，则执行 default 后的语句。C 语言还提供了一种 break 语句，专用于跳出 switch 语句。break 语句只有关键字 break，没有参数。在后面还将详细介绍。

例如：

```
void main()
{
    int a;
    printf("input integer number: ");
    scanf("%d",&a);
    switch (a)
    {
        case 1:printf("Monday\n"); break;
        case 2:printf("Tuesday\n"); break;
        case 3:printf("Wednesday\n"); break;
        case 4:printf("Thursday\n"); break;
        case 5:printf("Friday\n"); break;
        case 6:printf("Saturday\n"); break;
        case 7:printf("Sunday\n"); break;
        default:printf("error\n"); break;
    }
}
```

本程序要求输入 1～7 中的一个数字，输出对应星期的英文单词；不在 1～7 之间，则输出“error”。可见此类多分支程序用 switch 语句可轻松实现。

在使用 switch 语句时还应注意以下几点：

- 在 case 后的各常量表达式的值不能相同，否则会出现错误。
- 在 case 后允许有多个语句，可以不用大括号“{}”括起来。
- 各 case 和 default 子句的先后顺序可以变动，而不会影响程序执行结果。
- default 子句可以省略不用。

#### 3.1.4.2 循环语句

计算机的基本特性之一就是具有重复执行一组语句的能力——循环能力。循环结构是程序中一种很重要的结构。其特点是：在给定条件成立时，反复执行某程序段，直到条件不成立为止。给定的条件称为循环条件，反复执行的程序段称为循环体。C 语言提供了多种循环语句，可以组成各种不同形式的循环结构。

**1. while语句**

while 语句的一般形式如下：

```
while (表达式)
{
    语句          /* 循环体 */
}
```

这里的表达式是 while 循环能够继续的条件，而语句部分则是循环体，是执行重复操作的部分。当表达式为真时，执行循环体内的语句。其特点是：先判断表达式，后执行语句。循环条件测试位于循环体的开头，要想执行重复操作首先必须进行循环条件测试，若条件不成立则重复操作一次也不能执行。

while 语句的语义是：计算表达式的值，当值为真（非零）时，执行循环体语句。举例如下：

```
int sum =0;
int i;
while (i<=10)
{
    sum += i;
    i++;
}
```

程序执行后，sum = 55。

通常情况下，在 while 循环体中，应该有使循环趋于结束的语句，否则循环将无休止地继续。当然有些特殊情况需要这种无限循环，比如在等待中断时。

使用 while 语句应注意以下几点：

- while 语句中的表达式一般是关系表达式或逻辑表达式，只要表达式的值为真（非零）即可继续循环。
- 循环体如包括有一个以上的语句，则必须用大括号“{}”括起来，组成复合语句。

- 应注意循环条件的选择以避免死循环。
- 允许 while 语句的循环体又是 while 语句，从而形成双重循环。

**2. do-while语句**

do-while 语句的一般形式为：

```
do
{
    语句          /* 循环体 */
}
while(表达式);
```

其中语句是循环体，表达式是循环条件。

do-while 语句的语义是：先执行循环体语句一次，再判别表达式的值，若为真（非零）则继续循环，否则终止循环。

do-while 语句和 while 语句的区别在于 do-while 是先执行后判断，因此 do-while 至少要执行一次循环体。而 while 是先判断后执行，如果条件不满足，则一次循环体语句也不执行。while 语句和 do-while 语句一般都可以相互改写。do-while 语句举例如下：

```
int sum =0;
int i;
do
{
    sum += i;
    i++;
}
while (i<=10);
```

程序执行后，sum = 55。

对于 do-while 语句还应注意以下几点：

- 在 if 语句和 while 语句中，表达式后面都不能加分号，而在 do-while 语句的表达式后面则必须加分号。
- do-while 语句也可以组成多重循环，而且也可以和 while 语句相互嵌套。
- 在 do 和 while 之间的循环体由多个语句组成时，也必须用大括号“{}”括起来组成一个复合语句。
- do-while 和 while 语句相互替换时，要注意修改循环控制条件。

**3. for语句**

C 语言中的 for 语句使用最为灵活，不仅可以用于循环次数已经确定的情况，而且可以用于循环次数不确定而只给出循环结束条件的情况。其一般形式为：

```
for(表达式 1；表达式 2；表达式 3)
{
```

```
    语句          /* 循环体 */
}
```

“表达式 1”通常用来给循环变量赋初值，一般是赋值表达式。也允许在 for 语句外给循环变量赋初值，此时可以省略该表达式。“表达式 2”通常是循环条件，一般为关系表达式或逻辑表达式。“表达式 3”通常可用来修改循环变量的值，一般是赋值语句。这 3 个表达式都可以是逗号表达式，即每个表达式都可由多个表达式组成。3 个表达式都是任选项，都可以省略。

for 语句的语义是：首先计算“表达式 1”的值；再计算“表达式 2”的值，若值为真（非零）则执行循环体一次，否则跳出循环；然后再计算“表达式 3”的值，转回第 2 步重复执行。在整个 for 循环过程中，“表达式 1”只计算一次，“表达式 2”和“表达式 3”则可能计算多次。循环体可能多次执行，也可能一次都不执行。for 语句的示例如下：

```
void main()
{
    int n,s=0;
    for(n=1;n<=100;n++)
        s=s+n;
    printf("s=%d\n",s);
}
```

用 for 语句计算 s=1+2+3+……+99+100 显然带来了很大的方便。本例 for 语句中的表达式 3 为 n++，实际上也是一种赋值语句，相当于 n=n+1，以改变循环变量的值。

用 for 语句实现从 0 开始输出 50 个连续偶数的例子如下：

```
void main()
{
    int a=0;
    int n=50;
    for(;n>0;a++,n--)
    printf("%d ",a*2);
}
```

在本例的 for 语句中，表达式 1 已省去，表达式 3 是一个逗号表达式，由 a++和 n--两个表达式组成。每循环一次 a 自增 1，n 自减 1。a 的变化使输出的偶数递增，n 的变化控制循环次数。

在使用 for 语句中要注意以下几点：

- for 语句中的各表达式都可省略，但分号间隔符不能少。
- 在循环变量已赋初值时，可省去表达式 1。如省去表达式 2 或表达式 3 则将造成无限循环，这时应在循环体内设法结束循环。
- 循环体可以是空语句。
- for 语句也可与 while、do-while 语句相互嵌套，构成多重循环。

### 3.1.4.3　转移语句

程序中的语句通常总是按顺序或按语句功能所定义的方向执行的。如果需要改变程序的正常流向，可以使用转移语句。

在 C 语言中提供了 4 种转移语句：goto、break、continue 和 return。其中 return 语句只能出现在被调函数中，用于返回主调函数。下面主要介绍前 3 种转移语句。

**1. goto语句**

goto 语句也称为无条件转移语句，其一般格式如下：

```
goto 语句标号;
```

其中“语句标号”是按标识符规定书写的符号，放在某一语句行的前面，标号后加冒号（:）。语句标号起标识语句的作用，与 goto 语句配合使用。

例如：

```
label: i++;
loop: while(x<7);
```

C 语言不限制程序中使用标号的次数，但各标号不得重名。goto 语句的语义是改变程序流向，转去执行语句标号所标识的语句。goto 语句通常与条件语句配合使用，可用来实现条件转移、构成循环、跳出循环体等功能。但是，在结构化程序设计中一般不主张使用 goto 语句，以免造成程序流程的混乱，使理解和调试程序都产生困难。

前文统计键盘输入一行字符个数的例子可用 goto 语句实现如下：

```
#include"stdio.h"
void main()
{
    int n=0;
    printf("input a string\n");
    loop: if(getchar()!='\n')
    {
        n++;
        goto loop;
    }
    printf("%d",n);
}
```

本例用 if 语句和 goto 语句构成循环结构。当输入字符不为'\n'时即执行 n++进行计数，然后转移至 if 语句循环执行，直至输入字符为'\n'才停止循环。

**2. break语句**

break 语句只能用在 switch 语句或循环语句中，其作用是跳出 switch 语句或跳出本层循环，转去执行后面的程序。由于 break 语句的转移方向是明确的，所以不需要语句标号与之配合。break 语句的一般形式为：

```
break;
```

使用 break 语句可以使循环语句有多个出口，在一些场合下使编程更加灵活、方便。break 语句的使用可参见前面 switch 语句的例子。

### 3. continue 语句

continue 语句只能用在循环体中，其一般格式是：

```
continue;
```

其语义是：结束本次循环，即不再执行循环体中 continue 语句之后的语句，转入下一次循环条件的判断与执行。continue 语句和 break 语句的区别是：continue 语句只结束本次循环，而不终止整个循环的执行；而 break 语句则是结束整个循环过程，不会再去判断循环条件是否满足。例如：

```
void main()
{
    int n;
    for(n=6;n<=96;n++)
    {
        if (n%6!=0)
        continue;
        printf("%d ",n);
    }
}
```

本例的功能是输出 96 以内能被 6 整除的数。本例中，对 6～96 的每一个数进行测试，如该数不能被 6 整除，即模运算值不为零，则由 continue 语句转去下一次循环。只有模运算值为零时，才能执行后面的 printf 语句，输出能被 6 整除的数。

## 3.2 HCS08 单片机 C 语言的扩展语法

HCS08 C 语言的语法大致与标准 C 相同。前面介绍了有关标准 C 语言的语法，下面主要介绍 HCS08 C 语言对标准 C 语言的扩展语法。

### 3.2.1 基本数据类型

基本数据类型如表 3-3 所示。

表 3-3 基本数据类型列表

| 数据类型 | 大小 | 无符号（unsigned）数据范围 | 有符号（signed）数据范围 |
|---|---|---|---|
| char | 8 bits | 0~255 | –128~127 |
| short int | 16 bits | 0~65535 | –32768~32767 |
| int | 16 bits | 0~65535 | –32768~32767 |
| long int | 32 bits | 0~4294967295 | –2147483648~2147483647 |

ANSI C 没有为 int 类型规定长度，不过 8 位的单片机的 C 编译器一般都把 int 定义为 16 位有符号数。但是由于 8 位机处理 8 位数据是效率最高的，所以 int 类型和更大的类型都应该在必需要时才使用。最好不要使用浮点数。

char 类型可能在不同的编译器中有不同的定义，或者根本没有定义，为了便于程序的移植要求，最好不要用 char 这个符号，应给它起个别名。

可以为所有用到的数据类型重新进行定义：

```
typedef unsigned char       UINT8;
typedef signed char             SINT8;
typedef unsigned int        UINT16;
typedef int             SINT16;
typedef unsigned long int   UINT32;
typedef long int            SINT32;
```

使用这样的定义使程序的移植性较好。

为了节约有限的 RAM 资源，应该为每个变量尽量用最小的数据类型；尽量少用有符号数据；可以在表达式中用强制类型转换把数据长度缩到最短。

## 3.2.2　变量定义

普通变量的定义和访问同标准 C 语言一样，在 HCS08 C 语言中主要要解决映像寄存器变量和某些特殊变量的定位问题，即把这些变量存放在 RAM 中指定的位置。

**1. 映像寄存器定位**

映像寄存器是单片机中跟硬件有关的寄存器，它们都有各自固定的 RAM 地址，其定位有 3 种方法：

（1）宏定义

例如：#define PortA (*(volatile unsigned char * )0x0000 )

这样 PortA 成为一个地址在 0x0000 的 unsigned char 类型的变量。这个定义看起来很复杂，其实它也可以分解成几个很简单的部分来看。

( volatile unsigned char * )是 C 语言中的强制类型转换，它的作用是把 0x0000 这个纯粹的十六进制数转换成为一个（地址）指针，其中的 volatile 并不是必要的，它只是告诉编译器这个值与外界环境有关，不要对它优化。接下来在外面又加了一个*号，就表示 0x0000 内存单元中的内容了。经过这个宏定义之后，PortA 就可以被作为一个普通的变量来操作了，所有出现 PortA 的地方在编译的时候都被替换成( * ( volatile unsigned char * )0x0000 )，外面一层括号是为了保证里面的操作不会因为运算符优先级或者其他不可预测的原因被改变而无法得到预期的结果。

这种定义方法适合所有的 C 编译器，可移植性好，但 PortA 并不是一个真正的变量，只是一个宏名，当你调试一个程序的时候，无法在调试窗口观察它的值。另外连接器也失

去了灵活性，它得防止其他变量与此变量冲突。

（2）使用@关键字

例如: volatile unsigned char PortA　@0x0000;

@是编译器扩展的一个特殊修饰符，其他编译器很可能并不认识。这种定义具有很好的可读性，但失去了可移植性。

（3）使用段定义

这种方法分为 2 个步骤：首先把变量定义在段中，其次在连接参数文件（*.prm）中把段定位在一个合适的位置。

例如，第 1 步：在源程序文件中定义如下：

```
#pragma DATA_SEG PORTB_SEG
volatile unsigned char PortA;
#pragma DATA_SEG DEFAULT
```

这样变量 PortA 定义在段 PORTB_SEG 中。

第 2 步：在 prm 文件中定义如下：

```
SECTIONS
PORTB_SEG = READ_WRITE 0x0000 SIZE 1;
```

这样段 PORTB_SEG 定位在地址 0x0000 上。

这种方法可移植性也很差，如果你要把它移植到别的编译器上，将不得不修改源程序。

### 3.2.3 变量定义修饰符

变量定义有 3 个修饰符值得注意，虽然它们与标准 C 是相同的，但是在嵌入式 C 语言中又有不同的含义。

（1）static

在子函数中用 static 声明的变量是局部变量，但是退出这个子函数后其值不消失。下一次调用这个函数时仍可以访问到原来的值。注意，在子函数中声明的 static 变量只对声明它的函数可见，别的函数是不可以使用的。如果 static 变量是在模块中声明的，那么只有本模块的函数可以使用它，别的模块中的函数是不能访问的。

```
void MyFunction (void)
{
    static char myVar = 0;    //用 static 声明的局部变量
    myVar = myVar + 1;
}
void main (void)
{
     MyFunction();        //调用之前 myVar = 0，调用之后 myVar = 1
     MyFunction();        //调用之前 myVar = 1，调用之后 myVar = 2
}
```

（2）volatile

如果一个变量的值可能会被程序操作之外的其他操作所改变，那么你必须用 volatile 声明。在嵌入式系统中的其他操作是：中断服务程序的操作和硬件动作的操作。

用 volatile 声明的变量是不会被编译器优化掉的，如：

```
volatile unsigned char PortA          @0x0000;
```

PORTA 作为一个输入端口，其值是由外部设备决定的，由于外部设备的变化是随机的，因此第一次读取的值和第二次读取的值很可能不同，所以我们把它声明为 volatile 变量。

```
a = PORTA;
a = PORTA;
```

由于 PORTA 是用 volatile 声明的变量，编译器不会把它优化成一句，而如果不是用 volatile 声明的，编译器就会将第二句优化掉，从而使程序忽略输入端口的变化。

通常把嵌入式设备的所有外围器件寄存器都声明为 volatile 。

（3）const

修饰符 const 可以用在任何变量之前，告诉编译器把此变量存储在 ROM 中。ROM_VAR 段是定位 const 变量的默认段。

语法格式：#pragma CONST_SEG <段名>

例如：

```
#pragma DATA_SEG DEFAULT
#pragma CONST_SEG DEFAULT
static int a;//变量 a 存放在默认的 RAM 段 DEFAULT_RAM 中，DEFAULT_RAM 是段名
static const int c0 = 10;//变量 c0 存放在默认的 ROM 段 ROM_VAR 中，ROM_VAR 是段名
```

此时编译器选项“－Cc 必需”是打开的。如果编译器选项“－Cc 必需”是关闭的，则变量 a 和 c0 都定位在 DEFAULT_RAM 中。

例如：

```
#pragma DATA_SEG MyVarSeg
#pragma CONST_SEG MyConstSeg
static int a; //变量 a 存放在段 MyVarSeg 中，MyVarSeg 是段名
static const int c0 = 10; //变量 c0 存放在段 MyConstSeg 中，MyConstSeg 是段名
```

此时编译器选项“－Cc 必需”是打开的。如果编译器选项“－Cc 必需”是关闭的，则变量 a 和 c0 都定位在 MyVarSeg 中。

### 3.2.4　全局变量和局部变量

全局变量为整个程序而定义，存在于整个程序运行期间。它们占用固定的 RAM 资源，因此除非在必需的情况下，否则不要轻易使用。局部变量为某个函数而定义，只在此函数运行的时候占用栈空间。局部变量实质上是函数运行所需要的一段 RAM 空间，因此函数

不运行时，它们不占用 RAM 资源。全局变量通常是为了给中断服务函数传递参数而定义的，建议定义时把它们定义在一个相对集中的 RAM 空间中。

例如：

```
#pragma DATA_SEG SCI_DATA /* 这条预处理指令之后的所有变量定义在 SCI_DATA 段 */
char senderBuffer[50];
char receiverBuffer[100];
……
#pragma DATA_SEG DEFAULT /*这条预处理指令之后的所有变量定义在默认段*/
```

在参数文件中必须指定 SCI_DATA 段。

### 3.2.5 位定义和访问

HCS08 C 语言采用直接位访问的方法来访问位，位的定义采用联合和结构数据类型来实现。

例如：

```
volatile union {
  struct {
    unsigned char MWPR:1;    /* MWPR is bit 0 in FEETST */
    unsigned char STRE:1;    /* STRE is bit 1 in FEETST */
    unsigned char VTCK:1;    /* VTCK is bit 2 in FEETST */
    unsigned char FDISVFP:1; /*FDISVFP is bit 3 in FEETST */
    unsigned char FENLV:1;   /* FENLV is bit 4 in FEETST */
    unsigned char HVT:1;     /* HVT is bit 5 in FEETST */
    unsigned char GADR:1;    /* GADR is bit 6 in FEETST */
    unsigned char FSTE:1;    /* FSTE is bit 7 in FEETST */
} FEETST_BITS;
unsigned char FEETST_BYTE; /* Alternate definition of the
port as a 8-bit variables. */
} FEETST_struct @0x00F6;

/* Define Symbol to access register FEETST */
#define FEETST FEETST_struct.FEETST_BYTE

/* Define Symbols to access single bits in FEETST */
#define MWPR FEETST_struct.FEETST_BITS.MWPR
#define STRE FEETST_struct.FEETST_BITS.STRE
#define VTCK FEETST_struct.FEETST_BITS.VTCK
#define FDISVFP FEETST_struct.FEETST_BITS.FDISVFP
#define FENLV FEETST_struct.FEETST_BITS.FENLV
#define HVT FEETST_struct.FEETST_BITS.HVT
#define GADR FEETST_struct.FEETST_BITS.GADR
#define FSTE FEETST_struct.FEETST_BITS.FSTE
```

这里的“:1”表示仅需要一个位，HCS08 会把它们包装在一起形成一个字节。这样我们就可以以字节方式或位方式访问整个寄存器和位。

例如：

```
FEETST =0X80;//字节方式
MWPR = 1;//位方式
VTCK=1; // BSET 2,FEETST1
FDISVFP=0; // BCLR 4,FEETST1
```

### 3.2.6　中断服务程序定义

在 HCS08 C 语言中的中断服务程序是用中断函数来实现的。中断函数既没有入口参数也没有返回值，但可以用全局变量来实现。中断函数的定义方法有 3 种。

（1）用预处理“#pragma TRAP_PROC”定义

这种定义方法分为两步：首先在源程序中定义中断函数，其次在参数文件中指定各中断函数在中断向量表中的地址，即在参数文件中指定一个地址，该地址的内容是中断服务程序的入口地址（向量）。

例如：

```
unsigned int intCount = 1;
#pragma TRAP_PROC
void IntFunc(void)
{
intCount++;
}
#pragma TRAP_PROC
void IntFunc2(void)
{
intCount--;
}
#pragma TRAP_PROC
void IntFunc3(void)
{
intCount=intCount*5;
}
```

#pragma TRAP_PROC 仅对紧跟着它的函数有效，通知编译器位于它下面的函数是中断函数，其返回指令是 RTI，而不是 RET，因此每个中断函数前面都必须有这个预处理。

在参数文件中要加入以下内容：

```
VECTOR ADDRESS 0xFFF0 IntFunc1 /* 0xFFF0 包含 IntFunc1 的地址 */
VECTOR ADDRESS 0xFFF2 IntFunc2 /* 0xFFF2 包含 IntFunc2 的地址*/
VECTOR ADDRESS 0xFFF4 IntFunc3 /* 0xFFF4 包含 IntFunc3 的地址 */
```

（2）用关键字“interrupt”定义

格式为：

```
interrupt <函数名>
/* 在参数文件中指定中断类型号 */
{
……/*代码*/
}
```

关键字 interrupt 通知编译器位于它后面的函数名是中断函数，同样也要在参数文件中指定各中断函数在中断向量表中的地址。

例如：

```
interrupt IntFunc2()
{
……
}
```

在参数文件中加入:

```
VECTOR ADRESS 0xFFF2 IntFunc2
```

（3）用关键字“interrupt”和中断向量号定义

格式为：

```
interrupt <中断向量号> <函数名>
{
……/*code*/
}
```

这种定义方法的关键字“interrupt”通知编译器位于它后面的函数名是中断函数，且通过中断向量号指定了各中断函数在中断向量表中的地址。这种方法就不再需要修改参数文件了，移植性较好。

中断向量号与中断向量表地址的对应关系如下：复位向量为 0 号，位于地址 0xFFFE，1 号紧跟着 0 号，位于地址 0xFFFC，其余以此类推。

另外用预处理命令#pragma TRAP_PROC: SAVE_REGS 可以确保在中断函数中，所有 CPU 寄存器或编译器使用的伪寄存器内容不会被中断函数破坏。

### 3.2.7 内嵌汇编和混合调用

**1. 内嵌汇编**

HCS08 C 语言提供了一个很有用的功能——内嵌汇编。凡是 C 语言程序段能出现的地方汇编语言源程序都可以出现，但是汇编语言源程序必须位于一个 C 语言函数中。

语法格式如下：

① "asm" <汇编指令> ";" ["/*" 注释 "*/"]

双引号中的内容为关键字，方括号中的内容为可选项。

例如：

```
asm sta COPCTL;  /* 喂狗，即写看门狗寄存器，防止超时导致系统复位*/
```

② "asm" "{"
　　<汇编指令 1> [";" 注释]
<汇编指令 2> [";" 注释]
"}"

要求：大括号内每条汇编指令占一行；标号以“:”结尾占一行；注释以“;”开始；可以用变量名访问全局变量和 C 函数中的局部变量。汇编语言结束前要保证堆栈内容与汇编开始前一致。

例如：用内嵌汇编方法实现统计字符串中字符的个数。

```
int strlen (char *str)
/*** 'str' 参数由堆栈传递。函数返回字符串'str'的长度
假定字符串的长度小于 256!
*/
{
asm {
    LDHX str      ; 装入指针
    CLRA           ; 初始化计数器
    BRA test      ; 跳到 test
loop:
    AIX #1        ; 指针加 1
    INCA          ; 计数器加 1
test:
    TST 0,X      ; 字符串是否结束?
    BNE loop     ; 下一字符
    CLRX          ; 返回值在 X:A(参见后文)
    };
/* 这里可以继续写 C 语言程序段 */
}
```

**2. 混和调用**

混和调用指的是 C 语言函数调用汇编语言函数和汇编语言函数调用 C 语言函数。实现混和调用要解决两个问题，一是常量和变量的相互访问；二是调用协议。

（1）常量和变量的相互访问

C 语言函数访问汇编语言常量和变量。

在汇编语言源程序中采用汇编语法定义变量和常量；在 C 语言中把这些变量、常量声明为外部变量、常量，即可以用变量、常量名来访问。汇编器和编译器对段名的大小写敏感。

例如：

汇编源程序中：

```
XDEF ASMData, ASMConst ;声明 ASMData, ASMConst 可以在模块外访问
MyData: SECTION
ASMData: DS.W 1 ; 定义变量
MyConst: SECTION
ASMConst: DC.W $44A6 ; 定义常量
```

在 C 语言源程序中：

```
#pragma DATA_SEG MyData /* 定义变量段 */
extern int ASMData; /* 变量 ASMData 是外部模块声明的 */
#pragma DATA_SEG DEFAULT /* 返回默认变量段 */
#pragma CONST_SEG MyConst /* 定义常数段 */
extern const int ASMConst; /* 常量 ASMConst 是外部模块声明的*/
#pragma CONST_SEG DEFAULT /*返回默认常数段  */
```

汇编语言访问 C 语言变量、常量。

在 C 语言中用标准格式定义常量、变量，在汇编语言中声明这些变量、常量是外部定义的即可。

例如：

在 C 语言源文件中：

```
unsigned int CData; /* 定义变量 */
unsigned const int CConst; /* 定义常量 */
```

在汇编语言源文件中：

```
XREF CData ; 外部声明的变量
XREF CConst; 外部声明的常量
```

使用：

```
LDD CConst
STD CData
```

（2）调用协议

调用协议即函数参数传递和返回值的协议。对于 HC08 和 HCS08 来说调用协议是不同的。

HC08 参数传递：调用者函数把所需参数从左到右依次入栈，调用结束后由调用者函数从堆栈中清除传递的参数。在函数传递的是值参数的情况下，如果最后一个参数长度为 2 字节，则用 X:A 寄存器对传递；如果最后一个参数长度为 1 字节且倒数第 2 个参数长度超过 1 字节，则最后一个参数用寄存器 A 传递；如果最后一个参数长度为 1 字节且倒数第 2 个参数长度也是 1 字节，则最后一个参数用寄存器 X 传递，倒数第 2 个参数用寄存器 A 传递。

## 3.3　本章总结

在单片机程序设计中，C 语言已经成为首选的程序设计语言。采用 C 语言开发可以提高软件开发的工作效率，并提高程序的可靠性、可读性和可移植性。本章详细介绍了 C 语言设计基础和 HCS08 单片机的 C 语言扩展特性，重点包括：数据类型、变量定义、变量定义修饰符、全局变量和局部变量、位定义和访问、中断服务程序定义和内嵌汇编等。通过本章的学习，读者将了解和掌握 HCS08 系列单片机 C 语言程序设计的方法。

# 第 4 章

# HCS08 单片机的开发工具

学习 HCS08 单片机的应用开发，掌握其常用的开发工具十分重要。本章重点对 Freescale 单片机的 CodeWarrior 系列集成开发环境进行介绍。

CodeWarrior 系列集成开发环境 IDE（Integrated Development Environment）是 Metrowerks 公司为开发嵌入式微处理器而设计的一套强大易用的开发工具，使用它可以有效地提高软件开发效率。Freescale 公司的官方网站（www.freescale.com）免费提供下载多种版本，包括评估板、特别版、标准版和专业版等。

本章以 CodeWarrior for Microcontrollers V6.2 版本为例来讲解 CodeWarrior 的安装和使用方法。读者通过学习，将学会在 CodeWarrior IDE 中用汇编语言和 C 语言来实现对 HCS08 系列单片机的编程。

## 4.1 CodeWarrior for Microcontrollers V6.2 的安装

### 4.1.1 系统需求

基于 Microsoft Windows 操作系统平台的 CodeWarrior IDE 至少需要如下的计算机配置：

- 200MHz Pentium II 或 AMD-K6 处理器
- 128MB 的 RAM
- 最小模式安装需要大约 300MB 的硬盘空间
- 完全安装需要大约 700MB 的硬盘空间

- 需要 Microsoft Windows 98/ NT/ 2000/ XP 等操作系统
- 外设需有光盘驱动器 CD-ROM（不是必需的）、串口、并口或 USB 接口

### 4.1.2　CodeWarrior 的安装与卸载

**1. CodeWarrior的安装**

CodeWarrior IDE 的安装需要有完整的安装文件，可以通过光盘或硬盘来安装。下面以硬盘安装为例，简单地把安装过程介绍一下：

第一步，双击可执行文件“CW_V5_1.exe”运行安装程序。

第二步，单击【Next】按钮，选择读取安装程序后要放置的目录，单击【Next】按钮。

第三步，待读取自动完成后，单击【Next】按钮。

第四步，选择接受协议，连续两次单击【Next】按钮后，选择安装目录。

第五步，在两次单击【Next】按钮以后，单击【Install】按钮，CodeWarrior 的安装就开始了。

第六步，自动安装完成，取消复选框，单击【Finish】按钮，则安装完成（也可以选中复选框来选择自己想要的更新）。

第七步，打开集成开发环境。依次选择计算机中的“开始”→“程序”→“Freescale CodeWarrior”→“CW for Microcontrollers V6.2” →“CodeWarrior IDE”菜单。

**2. CodeWarrior的卸载**

如果您不再需要 CodeWarrior IDE，可以把它从您的计算机上卸载。通过操作系统自带的控制面板中的“添加/删除程序”进入卸载程序，单击【Remove】按钮，然后再单击【Next】按钮就可以自动完成卸载过程。单击【Modify】按钮可以添加或者删除功能组件，选择【Repair】可以在 CodeWarrior 软件崩溃后来修复它。

**3. CodeWarrior IDE的安装目录**

CodeWarrior IDE 的安装目录如表 4-1 所示。

表 4-1　CodeWarrior IDE 的安装目录

| 目 录 名 | 内　　容 |
|---|---|
| (CodeWarrior_Examples) | 用户程序实例 |
| (Helper Apps) | 辅助的应用程序 |
| Bin | IDE 的可执行文件及所有的 DLL 插件 |
| Help | CodeWarrior 的帮助文件 |
| Lib | HC08 系列处理器和 xgate 相关的各种库文件 |
| Lint | PClint 的安装目录和配置文件 |
| ProcessorExpert | 与处理器专家相关的文件 |
| prog | 编译、调试和仿真程序等 |
| Release_Notes | 一些关于开发环境的说明文件 |
| Stationery | 工程模板 |
| Templates | 默认的文档模板 |

# 4.2 CodeWarrior for Microcontrollers V6.2 功能使用

本节主要介绍 CodeWarrior IDE 的基本功能和使用方法。首先介绍如何进行功能设置。

## 4.2.1 集成环境与功能设置

### 1. 工具栏

如图 4-1 所示，工具栏的每个图标都对应一个独立的功能，所有的功能都可以在对应的菜单栏里找到。



图 4-1 CodeWarrior IDE 的工具栏

### 2. 菜单栏

如图 4-2 所示，CodeWarrior 的菜单栏里基本包含了所有功能，在工具栏里有的，都能在菜单栏里面找到相应的命令。

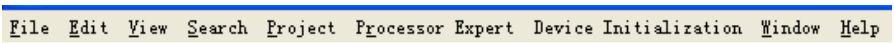



图 4-2 CodeWarrior IDE 的菜单栏

### 3. CodeWarrior IDE的设置

选择【Edit】→【Preferences...】命令来设置 CodeWarrior IDE 的有关参数，出现如图 4-3 所示的 CodeWarrior IDE 的参数设置对话框，用户可按照自己的要求来定制开发环境。

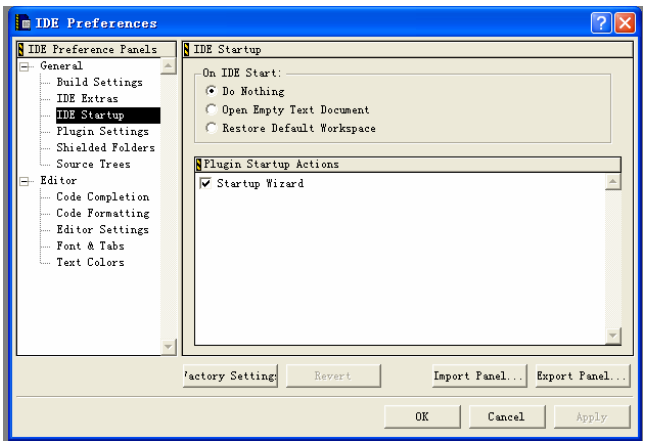



图 4-3 CodeWarrior IDE 的参数设置对话框

图 4-3 里包括可以控制 CodeWarrior 的代码编辑器、调试工具和其他功能的界面和行为选项。

### 4. 工程窗口介绍

由 CodeWarrior 创建的工程包含多个文件，其中工程文件扩展名为 mcp，它包含了所有文件的指针和工程的配置信息。这些配置信息包括编译和链接的设置、源文件、库文件及它们最终产生可执行程序的依赖关系。可以说这个 mcp 文件是生成可执行文件的核心。我们通过打开 mcp 文件就可以打开一个工程，就可以管理工程内的所有文件和将要生成的目标文件。

每个工程可以有很多编译目标（Build Target），编译目标可以告诉 IDE 来产生怎样的输出文件。如图 4-4 所示，这里的编译目标就是 P&E Multilink/Cyclone BDM（使用 BDM 下载程序到单片机）；当然可以通过该下拉框，根据自己的需要来选择其他的目标。如果需要通过模拟器（Simulator）进行全片仿真的话，就要选择相对应的编译目标。

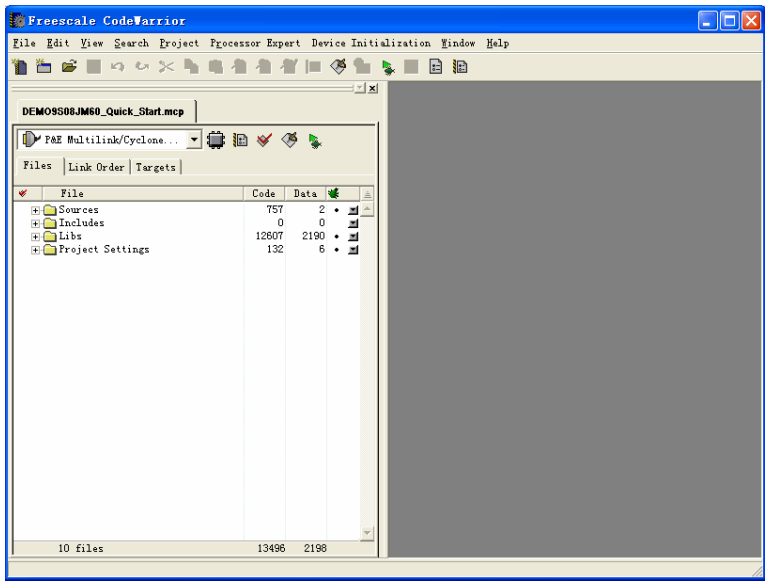



图 4-4　CodeWarrior IDE 的工程窗口

### 5. CodeWarrior IDE代码编辑器的使用

CodeWarrior IDE 的代码编辑器可以完成两个任务：一是建立和编辑程序源文件，这也是代码编辑器最主要的功能；二是建立和编辑文本文件。其实程序源文件的格式也是文本格式的，表面上看这两者似乎没有什么不同，但是作为两个独立的功能，它们还是有本质的差异的。程序源文件在编译时，是要参与编译并且生成目标文件的，而一般的文本文件，例如说明文件 readme.txt，是不参与编译的。在文件里也不包含任何程序代码和数据，我们可以轻松地利用代码编辑器来管理这些文本文件。

我们可以在工程窗口里任意单击一个程序源文件来打开代码编辑器窗口，此处我们打开主程序文件 DEMOJM_Quick_Start.c，如图 4-5 所示。

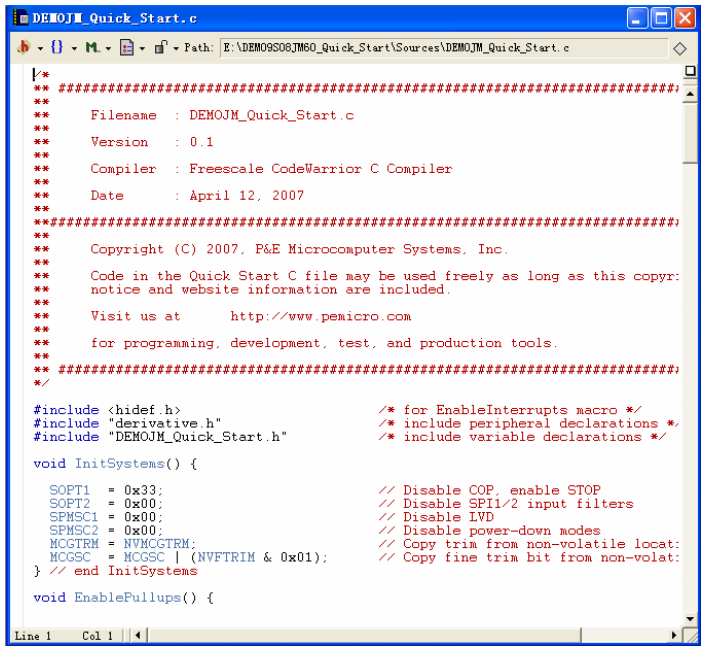



图 4-5　代码编辑器

代码编辑器的使用和常用的文本编辑器相似，这里就不再介绍了。

### 6. 编译目标的设置

在编译和链接前，可以针对编译目标进行设置。启动编译目标设置对话框，如图 4-6 所示，其中有很多种方法供用户选择，用户可以根据自己的需要灵活选择设置。

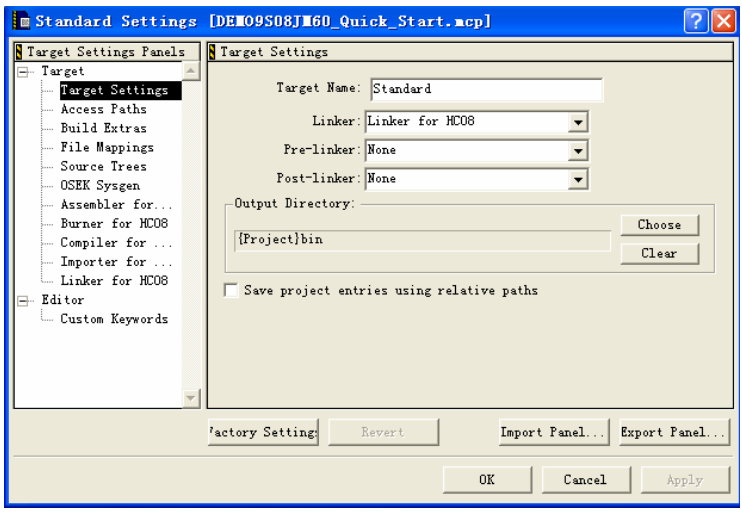



图 4-6　编译目标的设置

# 4.3　CodeWarrior 开发 HCS08 单片机的过程

为了使用 CodeWarrior IDE 来进行 HC(S)08 系列单片机程序的开发，我们必须通过创建很多且不同类型的文件来构成一个工程。这些不同类型的文件信息及其相互的依赖关系都被存储到一个扩展名为 mcp 的工程文件中，所以，打开这个 mcp 文件就相当于打开了一个完整的工程。

下面逐步讲解如何建立和打开工程；怎样快捷高效地编写程序源代码；工程建立后，如何编译和链接；最后详细介绍调试方法。

## 4.3.1　建立和打开工程

### 1. 建立一个新工程

下面结合一个实例来介绍如何新建一个工程。建立的工程取名为 prog_test。

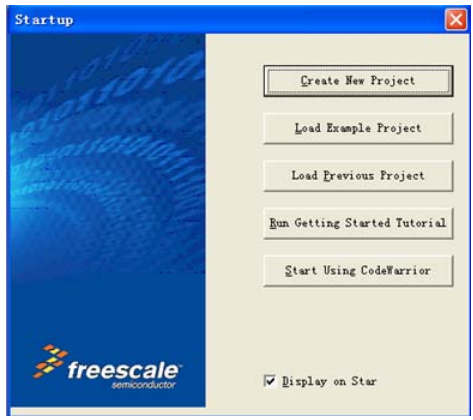



图 4-7　CodeWarrior V6 集成开发环境主界面

①启动 CodeWarrior IDE，即可打开 Codewarrior V6 集成开发环境主界面，如图 4-7 所示。中间有一个 Startup 对话框，第 1 个是创建新工程，第 2 个是装载实例工程，第 3 个是装载最近几次用过的工程，第 4 个是运行入门教程，第 5 个是开始使用 CodeWarrior。单击【Start Using CodeWarrior】按钮，进入 CodeWarrior 集成开发环境。

②在 CodeWarrior V6 集成开发环境的菜单栏中选择【File】→【New】命令或者按【Ctrl】+【Shift】+【N】组合键，启动新建对话框，如图 4-8 所示。或者在上一步中选择【Create New Project】按钮，弹出如图 4-8 所示的新建工程对话框。可以选择所用芯片的系列和型号，以及仿真调试模式。例如选择 HCS08→HCS08JM Family→MC9S08JM60，出现如图 4-9 所示的对话框。

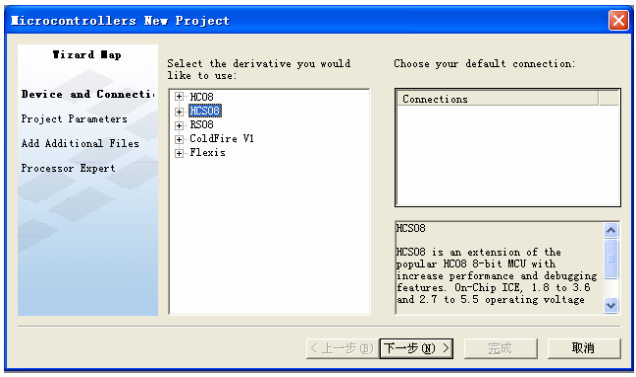



图 4-8　新建工程对话框

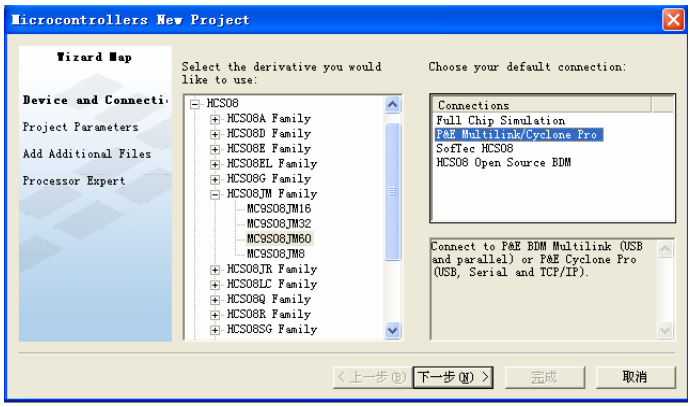



图 4-9　选择单片机的型号和仿真模式

如图 4-9 所示，仿真模式有 4 种。

- Full Chip Simulation 模式：允许单片机及其外围设备进行在线调试，或者利用软件进行模拟。
- P&E Multilink/Cyclone Pro 模式：当 HCS08 与外围设备硬件接口通过 P&E 提供时，要采用这种模式。
- SofTec HCS08 模式：当 HCS08 连接到基于 SofTec 微系统工具的任何一种 USB 时，要采用这种模式。
- HCS08 Open Source BDM 模式：当连接到基于 Freescale 的 HCS08 开源 BDM 电缆的 USB 时，要采用这种模式。

③下一步，紧接着会弹出如图 4-10 所示的对话框，在左侧选择工程中使用的语言，该开发环境支持的语言有 C、C++和可重定位汇编，可以选择其中一个，也可以复选，这里选择 C。在“Project name”文本框中输入工程名，在“Location”文本框中选择工程存放路径。

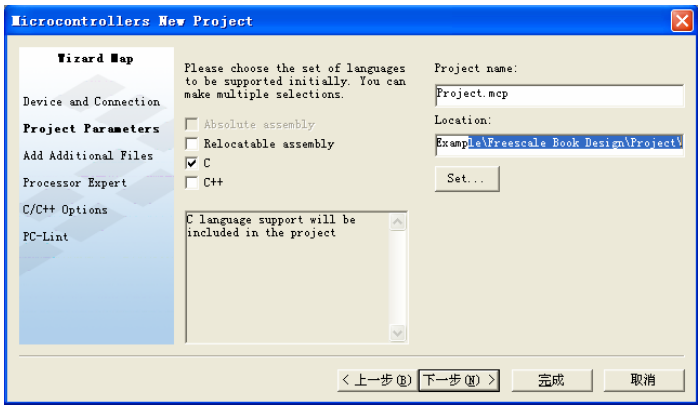



图 4-10　设置工程名称及路径

④进入下一步，紧接着会弹出如图 4-11 所示的对话框，这里我们可以添加已有的文件到工程下。这里没有文件，直接单击【下一步】按钮。

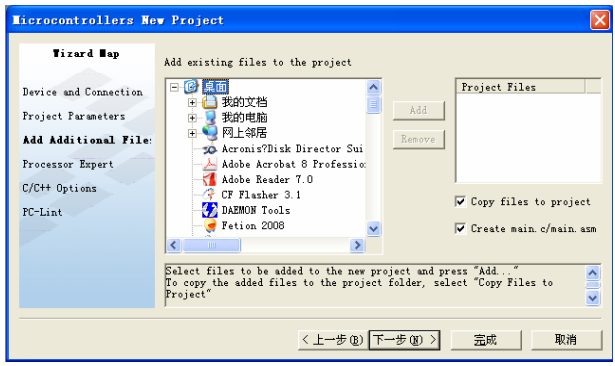



图 4-11　添加已有文件到工程

⑤紧接着会弹出如图 4-12 所示的对话框，可选择“Device Initialization”（器件初始化）和“Processor Expert”（处理器专家）快速应用开发工具自动生成初始化代码。

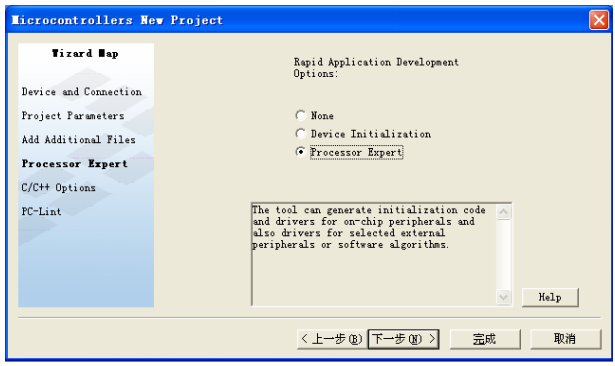



图 4-12　处理器专家设置

⑥进入下一步，紧接着会弹出如图 4-13 所示的对话框，这里有很多选项，如启动代码的类型、存储器模式和所支持的浮点数格式等。

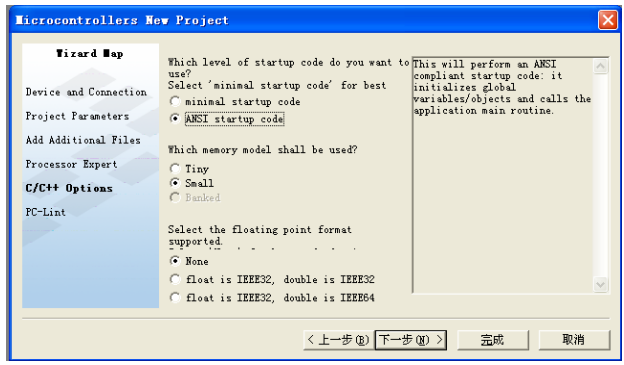



图 4-13　其他选项

⑦下一步紧接着会弹出如图 4-14 所示的对话框，选择是否创建一个 PC-lint(TM) 工程。

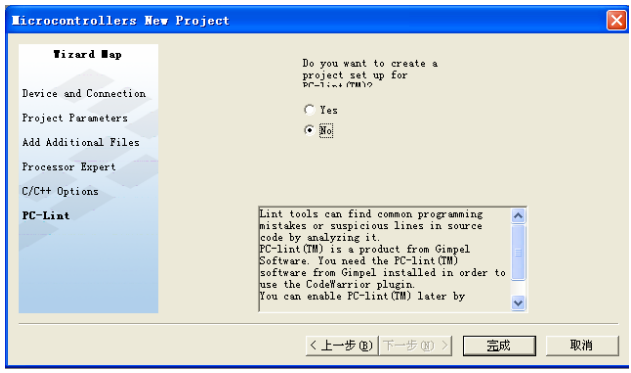



图 4-14　创建 PC-Lint(TM)工程

⑧单击【完成】按钮，工程创建完毕，若不需要一些附加功能，也可以在以上任一步骤中单击【完成】按钮结束工程创建。工程创建完成的窗口如图 4-15 所示。

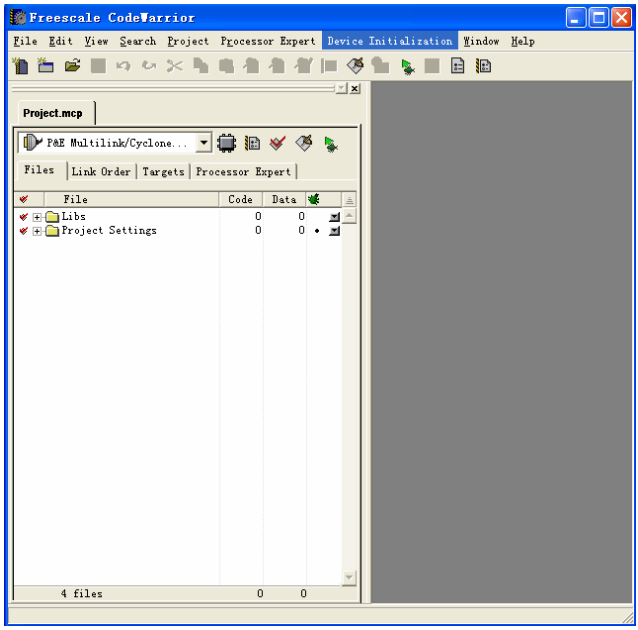



图 4-15　工程创建完成的窗口

**2. 打开一个工程**

以刚建立的 project 工程为例。打开该文件夹，然后双击 project.mcp 文件，这时 CodeWarrior 会自动打开，并且连 project 工程也一起打开了。

这里可能会遇到一个问题，就是我们在安装 CodeWarrior 时，Windows 也许没有自动把 mcp 文件与 CodeWarrior 关联，这样就不能通过双击某个 mcp 文件来打开一个工程。解决办法非常简单，用鼠标右键单击 mcp 文件，然后选择【打开方式...】命令，在弹出的对话框中选择“Integrated Development Environment”，或者通过浏览来找到安装后的

CodeWarrior IDE 程序，一般在 CodeWarrior 安装目录下的“bin\IDE.exe”，在单击【确定】按钮前选中“始终使用选择的程序打开这种文件”选项。当然也可以直接通过开始菜单中的快捷方式或者“bin\IDE.exe”打开 CodeWarrior，然后再打开工程。

### 4.3.2　编译和链接

下面仍以工程 project 为例。在 CodeWarrior IDE 中，我们可以直接用鼠标左键单击工程窗口中的【Make】按钮来完成编译和链接。这时，我们可以看到，在打了修改标记的文件前面会逐一出现动态的小齿轮，这表示正在编译这个文件。

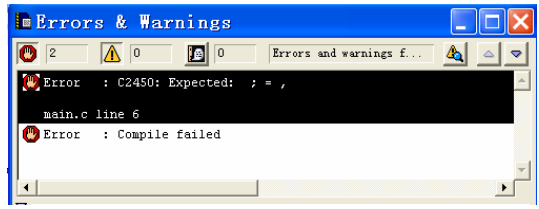



图 4-16　“Errors&Warnings”窗口界面

编译好后，会出现“Errors & Warnings”窗口，如图 4-16 所示。在这个窗口里显示的是错误和警告信息，其中，错误信息是必须消除的。在这些信息中包含的基本都是语法错误，而警告信息一般也需要排除，除非你能确认这个警告对于程序的结果没有影响。

### 4.3.3　软件仿真调试方法

编译工程，排除语法错误后，就开始调试程序了。本节将介绍调试程序的一般方法和步骤。调试程序和编译、链接不同，它依赖于程序员的经验，经验丰富的程序员就可以在较短的时间内调试程序达到自己的要求。所以，一定要多练习和实践，才能提高自己的调试效率。

首先要在工程的目标窗口中选择“Full Chip Simulation”，如图 4-17 所示。然后通过单击“Debug”快捷按钮，弹出如图 4-18 所示的仿真和实时调试窗口。




图 4-17　编译目标的选择

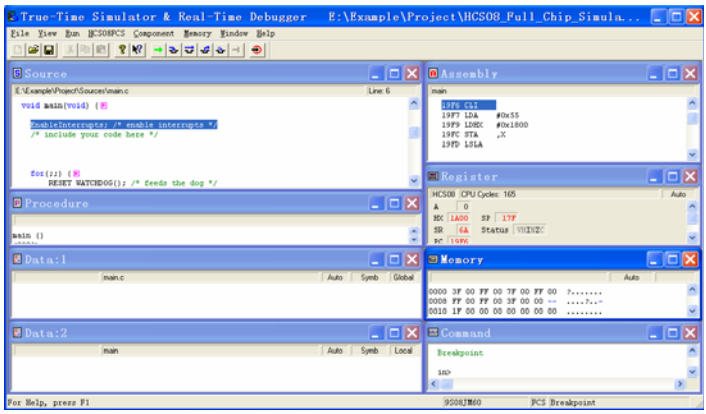



图 4-18　仿真和实时调试窗口

通过仿真和实时调试窗口可随时检查各寄存器、全局变量、局部变量和 RAM 中的内容。在图 4-18 中包括 Source、Procedure、Data1、Data2、Assembly、Register、Memory 和 Command 8 个子窗口。

Source 窗口即用户源程序窗口，单击鼠标右键弹出快捷菜单，在其中有几个重要的命令，如图 4-19 所示，功能分别如下：

- Set Breakpoint　　设置断点
- Run To Cursor　　运行到光标处
- Show Breakpoints　　显示断点
- Show Location　　显示地址
- Open Source File　　打开源文件
- Go To Line　　跳转行

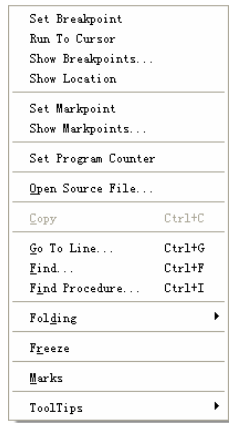



图 4-19　Source 窗口

无论是脱机仿真还是在线调试，均可利用快捷菜单任意设置一个或多个断点，方便地检查程序运行到断点处时各寄存器及状态变量的内容。

在图 4-18 中，Data1 数据窗口用于显示用户在源程序中定义的全局变量，还有牵涉到的相关寄存器设置。Data2 窗口显示的是源程序中的子函数变量（局部变量），单步运行时即可在这两个窗口中查看用户自定义变量的值是否正确。

Command 窗口显示程序运行过程中所走过的命令。Assembly 反汇编窗口中，蓝条指示的位置就是程序下一条要执行的指令。Register 窗口如图 4-20 所示。

在图 4-20 中，左上角显示 CPU 类型是 HCS08，紧接着“CPU Cycles:”代表 CPU 运行的指令周期，执行程序时，可以看到“CPU Cycles”后面的数值不断变化，从而可以精确计算 CPU 的执行时间。

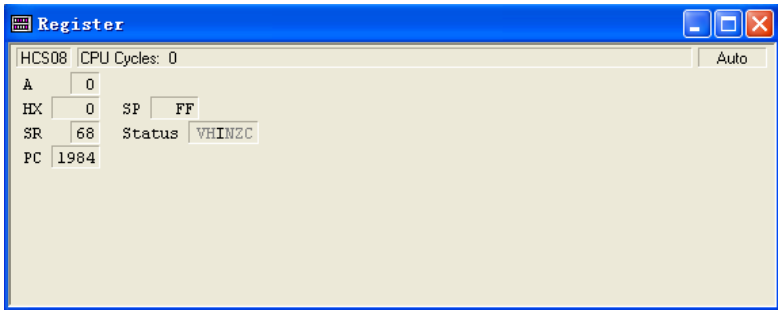



图 4-20　Register 窗口

因此，通过 CodeWarrior V6 可以很方便地实现程序下载、单步调试等，为系统程序的开发提供了非常好的仿真平台。

### 4.3.4　BDM 调试方法

当程序在“Full Chip Simulation”下调试通过后，就可以把程序下载到单片机内部进行在线调试了。

在线调试所需的仿真器有“P&E Multilink/Cyclone Pro” 和“HCS08 Open Source BDM”。前者目标板（实验电路板）电源由 BDM 提供；后者目标板可以由外部单独供电，也可以由 BDM 供电。在线调试的步骤如下：

第一步，建立工程。

第二步，编写程序。用户代码的位置和如何定位程序见前文介绍。

第三步，调试。

程序修改或编写完成后，选用“P&E Multilink/Cyclone Pro”或“HCS08 Open Source BDM”模式。在此我们选用后者，单击【Make】按钮进行编译链接。若编译链接无错误，则选择【Project】→【Debug】命令或工具栏内的【Debug】按钮，出现了如图 4-21 所示的对话框，进行程序下载连接。

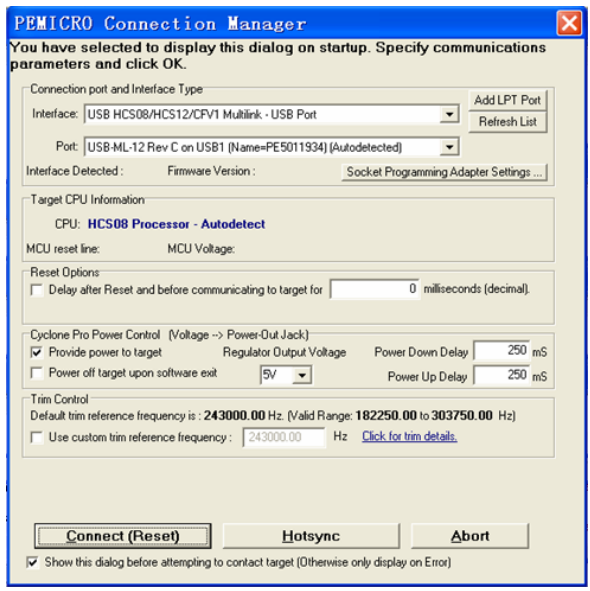



图 4-21　程序下载连接

在图 4-21 中单击【Connect】按钮，弹出如图 4-22 所示的对话框，最后单击【Yes】按钮，程序就下载到目标板了，并进入到如图 4-23 所示的调试窗口。

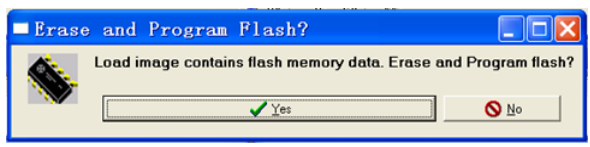



图 4-22　FLASH 擦除对话框图

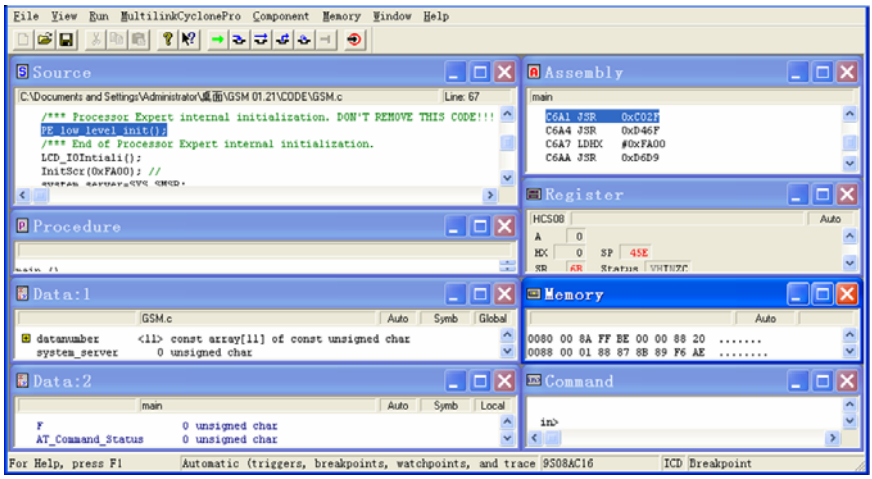



图 4-23　调试窗口

## 4.4　本章总结

本章详细介绍了 HCS08 单片机的集成开发环境 CodeWarrior V6.2，主要包括 CodeWarrior V6.2 的安装、操作和调试，其中调试部分分别介绍了软件仿真调试和硬件仿真调试，并给出了详细的操作步骤。通过本章的学习，读者可以熟悉 CodeWarrior V6.2 集成开发环境的各种操作及使用方法。

# 第二篇

# 常用模块设计实例篇


- 第 5 章　通用 IO 模块：流水灯设计实例
- 第 6 章　通用 IO 模块：语音播放系统设计实例
- 第 7 章　液晶显示模块：温度测量系统设计实例
- 第 8 章　液晶显示模块：实时时钟设计实例
- 第 9 章　KBI 模块：扫描键盘设计实例
- 第 10 章　定时器模块：用定时器控制蜂鸣器发声
- 第 11 章　定时器模块：电子钟系统设计实例
- 第 12 章　存储器模块：FLASH 编程实现
- 第 13 章　AD 模块：加速度传感器系统设计实例
- 第 14 章　AD 模块：烟雾检测系统设计实例
- 第 15 章　PWM 模块：寻迹智能小车控制系统设计实例
- 第 16 章　通用串口通信模块：单片机与 PC 通信设计实例
- 第 17 章　RS-485 总线通信模块：现场监测系统设计实例
- 第 18 章　$I^2C$ 总线通信模块：串行存储器接口设计实例
- 第 19 章　$I^2C$ 总线通信模块：$I^2C$ 主从通信系统设计实例
- 第 20 章　SPI 总线通信模块：电量监测仪设计实例
- 第 21 章　SPI 总线通信模块：SD 卡数据存储系统设计实例
- 第 22 章　USB 串行总线模块：数据传输系统设计实例

# 第 5 章

# 通用 IO 模块：流水灯设计实例

在单片机应用中，人机接口是非常重要的组成部分，如显示系统的工作状态等。在这些应用中，LED 发光二极管是一个不可缺少的部分。采用 MC9S08JM60 单片机很容易实现流水灯的设计。本章从硬件和软件两个方面介绍 MC9S08JM60 的流水灯设计。

## 5.1 实例内容说明

在 HCS08 系列单片机中，所有 I/O 端口的管脚都是双向的，即可以实现输入与输出的控制。设置 I/O 端口相应的寄存器就可以实现相应的操作（如读/写、中断等）。I/O 端口相关的寄存器主要有：PTXD、PTXDD、PTXPE、PTXSE、PTXDS 等（其中 X 代表 A～G）。端口寄存器都是 8 位的寄存器，寄存器的每一位都对应一个 I/O 管脚，设置该位就可以设置该管脚的相应功能。下面为端口 A 的数据方向寄存器的示意。

| 数据位 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|
| 定义 | 0 | 0 | PTADD5 | PTADD4 | PTADD3 | PTADD2 | PTADD1 | PTADD0 |
| 复位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

端口的其他寄存器的结构和上面的结构类似，都是每一位对应一个 I/O 管脚，这里不再对所有的端口寄存器进行介绍。

由于端口的每个管脚可以独立设置，并且能分别设置成输入与输出方向，因此利用 I/O 管脚就很容易实现流水灯的设计。

## 5.2　设计思路分析

本例流水灯设计选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60，其内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、I2C 控制器、定时器等多种外设，方便进行系统设计。

流水灯系统的设计比较简单，选定单片机后，辅之以必要的外围电路，即可构成系统。

## 5.3　硬件电路设计

流水灯的硬件电路比较简单，主要是 LED 电路。由于电源电路、复位电路和 BDM 仿真调试电路贯穿整本书，因此也在这里进行介绍。在后续的章节中，如果系统的电源要求一致，则限于篇幅，对电源电路、复位电路和 BDM 仿真调试电路将不再进行介绍。

### 5.3.1　电源电路

由于 MC9S08JM60 单片机的工作电压为 2.7V～5.5V，考虑到各种应用场合下，系统的功耗可能相差较大，我们选用 AS1117 作为电源芯片。该电源芯片的输出电压为 3.3V，电流为 500mA，完全能满足大多数应用场合的需求。如图 5-1 所示为具体的电路图。

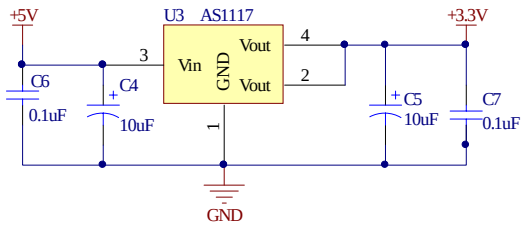



图 5-1　电源电路

由上图可以看出：该电源电路非常简单，只需要简单的外围器件。为了减少电源的噪声干扰，芯片的输入端和输出端都加上了旁路电容和去耦电容进行滤波处理。

### 5.3.2　复位电路

在单片机系统中，为了保证系统在上电时进行可靠的初始化，同时也为了保证对电源的监视，需要采用复位电路。常用的复位电路有 RC 复位电路和复位芯片，在此我们采用复位芯片构成复位电路。如图 5-2 所示为复位电路。

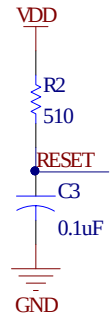



图 5-2　复位电路

### 5.3.3 BDM 仿真调试电路

MC9S08JM60 单片机内置 BDM 仿真调试电路，外围只需一个简单的 BDM 接口实现单片机的仿真调试。如图 5-3 所示为 BDM 仿真调试电路。

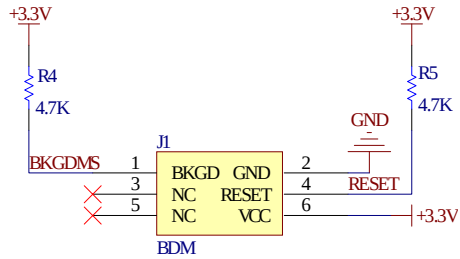



图 5-3 BDM 仿真调试电路

由上图可以看出，仿真调试电路非常简单，只需两根信号线、电源和地信号即可。

### 5.3.4 流水灯电路

流水灯电路主要是利用 MC9S08JM60 单片机的一般 I/O 端口来进行设计的。流水灯由 8 个 LED 组成，以 200ms 的速度显示，表示二进制灯。如图 5-4 所示为流水灯的电路设计图。

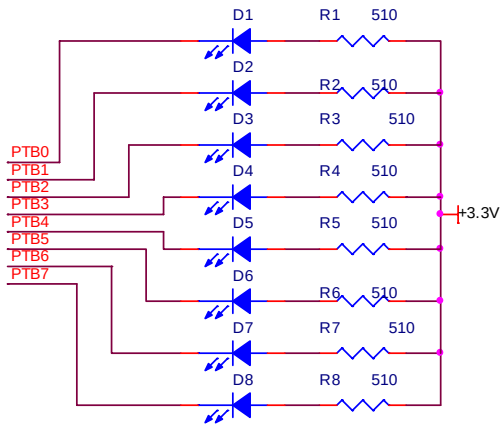



图 5-4 流水灯电路

在上图中，端口 B 的 8 个 I/O 管脚为流水灯的控制线，配置为输出端口。当端口 B 的相应 I/O 管脚输出低电平，则对应的 LED 被点亮，通过控制点亮的次序和时间，即可实现二进制流水灯显示。

### 5.3.5 单片机电路

单片机电路非常简单，只需要单片机的端口 B 与流水灯电路接口，如图 5-5 所示。其中流水灯电路主要是利用 MC9S08JM60 单片机的一般 I/O 端口来进行设计的，流水灯由 8

个 LED 组成，以 200ms 的速度显示。

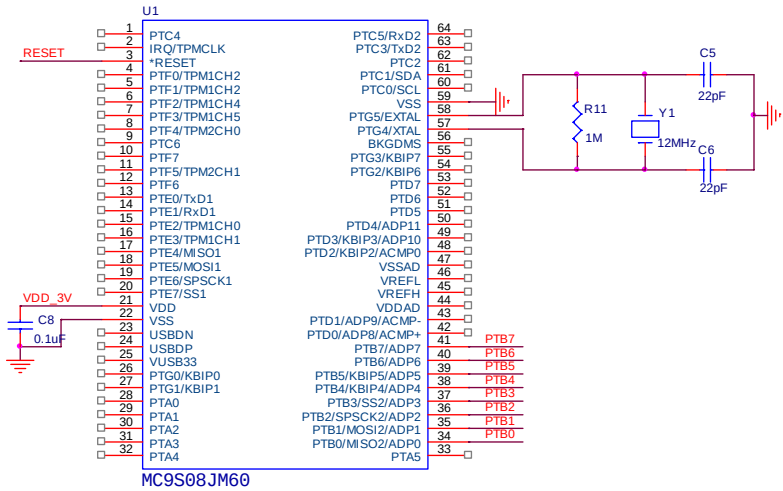



图 5-5　单片机电路

# 5.4　软件设计与代码

## 5.4.1　软件设计流程

流水灯系统软件采用前后台方式。系统初始化完成后，即进入主循环，二进制显示流水灯。系统初始化主要包括多功能时钟发生器 MCG 的初始化和 I/O 管脚的初始化。下面给出主程序的具体程序。

## 5.4.2　初始化程序

初始化程序主要包括 MCG 初始化、端口初始化两个部分，下面分别介绍具体的程序。

**1. MCG初始化**

MCG 模块属于仅写一次寄存器，在硬件上电复位后进行设置。具体程序如下：

```
void _EntryPoint(void)
{

  /* 系统选项寄存器 1SOPT1:设置 */
  setReg8(SOPT1, 0x13);
  /*系统电源管理状态和控制寄存器 1SPMSC1 设置 */
  setReg8(SPMSC1, 0x1C);
  /*系统电源管理状态和控制寄存器 2SPMSC2 设置 */
  setReg8(SPMSC2, 0x00);
  /* 系统时钟初始化*/
```

```
  MCGTRM = *(unsigned char*)0xFFAF;
MCGSC = *(unsigned char*)0xFFAE;
  /*MCG 控制寄存器 2（MCGC2）设置 */
  setReg8(MCGC2, 0x76);
  /*MCG 控制寄存器 1（MCGC1）设置 */
  setReg8(MCGC1, 0x3A);
  /*MCG 控制寄存器 3（MCGC3）设置 */
  setReg8(MCGC3, 0x01);
  /*等待外部参考时钟稳定 */
  while(!MCGSC_OSCINIT) {  }
 /*等待外部参考时钟被选定*/
  while(MCGSC_IREFST) {    }
/*等待 FLL 锁定*/
  while(!MCGSC_LOCK) {     }
/*等待 FLL 时钟被选定为总线参考时钟 */
 while((MCGSC & 0x0C) != 0x00) {   }
  /*跳转到 C 启动代码处 */
   __asm   jmp _Startup ;
}
```

**2. 端口初始化**

端口初始化程序主要设置相应的 I/O 管脚控制寄存器，以实现管脚的输入、输出和内部上拉等功能，在此我们将 PTB 口全部设置为输出，以驱动 8 个 LED。具体程序如下：

```
void PE_low_level_init(void)
{
    /*PTB 数据寄存器初始化*/
    setReg8(PTBD, 0x00);
     /* 端口 B 上拉电阻使能寄存器设置 */
setReg8(PTBPE, 0x00);
/*PTB 数据方向寄存器初始化*/
    setReg8(PTBDD, 0xFF);
    /* 端口输出转换速率控制寄存器设置 */
clrReg8Bits(PTASE, 0x3F);
    setReg8(PTBSE, 0x00);
    clrReg8Bits(PTCSE, 0x7F);
    setReg8(PTDSE, 0x00);
    setReg8(PTESE, 0x00);
    setReg8(PTFSE, 0x00);
    clrReg8Bits(PTGSE, 0x3F);
    /* 端口输出驱动能力选择寄存器设置 */
    setReg8(PTADS, 0x3F);
    setReg8(PTBDS, 0xFF);
    setReg8(PTCDS, 0x7F);
```

```
    setReg8(PTDDS, 0xFF);
    setReg8(PTEDS, 0xFF);
    setReg8(PTFDS, 0xFF);
    setReg8(PTGDS, 0x3F);
    /*使能系统中断*/
    __EI();
}
```

### 5.4.3　主程序

系统经过一系列初始化后，即进入主程序工作，主程序逐次点亮 LED，实现二进制流水灯显示。下面给出流水灯的主程序：

```
void main(void)
{
    unsigned char n;
    /*端口初始化*/
    PE_low_level_init();
    for(;;) {
         for(n=0;n<0xFF;n++)
        {
           /*指定端口 LED 点亮*/
            LED_ClrBit(n);
            Delay_Ms(200);
        }
  }
```

## 5.5　实例总结

本章通过利用单片机的 I/O 口，辅之以简单的外围电路，实现了二进制 LED 流水灯的设计。通过该实例，读者可举一反三，了解 I/O 端口使用的一般步骤，为后续进行复杂的系统设计打下良好的基础。

# 第 6 章

# 通用 IO 模块：语音播放系统设计实例

随着单片机技术的日益发展，人们已经不再满足于键盘输入、屏幕输出这样传统的输入输出方式，而希望拥有更好的人机界面、更方便的操作方式。具有语音功能的单片机系统于是应运而生，而且获得了广泛的应用，比如公交车报站器、出租车语音播报器等。本章从硬件和软件两个方面来介绍基于MC9S08JM60单片机通用I/O模块实现的语音播放系统的开发。

## 6.1 实例内容说明

在声学领域，单片机技术与各种语音芯片相结合，即可实现单片机语音系统。语音芯片，就是在人工或控制器的控制下可以录音和放音的芯片。本章将利用 MC9S08JM60 单片机和语音芯片，辅之以必要的外围器件，实现一个典型的单片机语音播放系统。如图 6-1 所示为系统的原理框图。

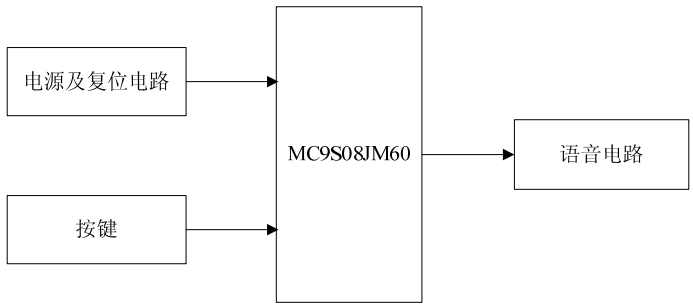



图 6-1　系统原理框图

如图 6-1 所示，语音播放系统的结构比较简单。按键负责语音的录音和播放，语音芯片的控制通过 MC9S08JM60 的 I/O 口实现，辅之电源、复位和 BDM 仿真电路，即可完成系统硬件平台的搭建。

## 6.2　设计思路分析

语音播放系统的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、I2C 控制器、定时器等多种外设，方便进行系统设计。

分析图 6-1 所确定的系统原理框图可知，语音芯片是整个系统的关键，因此首先需选择合适的语音芯片。

目前市场上流行的语音芯片有很多，从性价比的角度考虑，美国 ISD 公司的 ISD 系列语音芯片可谓一枝独秀。ISD 语音芯片具有以下优点：

- 采用多电平直接模拟量存储专利技术，即将模拟语音数据直接写入到单个存储单元，不需经过 A/D、D/A 转换，因此能够非常真实、自然地再现语音的自然效果，避免一般固体语音电路因量化和压缩所造成的量化噪声和失真现象。
- 内部集成了大容量的 EEPROM，无需再扩展存储器。
- 控制简单、控制引脚与 TTL 电平兼容。

因此，本例选用 ISD 公司的语音芯片 ISD2560。ISD2560 的录音时间是 60s。

## 6.3　硬件电路设计

语音播放系统主要由语音电路、电源电路、复位电路和单片机电路等组成。电源电路、复位电路和 BDM 仿真调试电路请读者参考第 5 章。本节主要介绍语音电路和单片机电路，在介绍具体的电路之前，首先对 ISD2560 进行简单的介绍。

### 6.3.1　ISD2560 介绍

ISD2560 是 ISD 公司 ISD2500 系列语音芯片中的一款，其主要功能特性如下：

- 录音时间可达 60s。
- 信息可无电保存 100 年。
- 重复录音最多可达 10 万次。
- 具有多段信息处理功能，最大可分 600 段。
- 自动节电模式，此时消耗电流仅为 1uA。
- +5V 供电。

为便于进行硬件电路的设计，下面给出该芯片的管脚图，如图 6-2 所示。

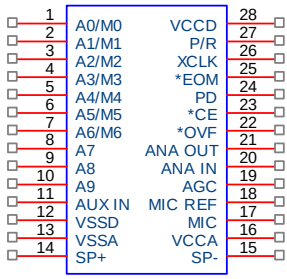



图 6-2　ISD2560 管脚图

由图 6-2 可知，该芯片有 28 个管脚，下面对具体的管脚进行介绍，见表 6-1。

表 6-1　ISD2560 管脚对应表

| 引 线 端 | 名　　称 | 功　　能 |
|---|---|---|
| 1~7 | A0/M0~A6/M6 | 地址 |
| 8~10 | A7~A9 | 地址 |
| 11 | AUX IN | 辅助输入 |
| 12，13 | VSSD、VSSA | 数字和模拟地址 |
| 14，15 | SP+、SP- | 扬声器输出 |
| 16，28 | VCCA、VCCD | 模拟、数字信号电源正极 |
| 17，18 | MIC、MIC REF | 麦克风输入和输入参考端 |
| 19 | AGC | 自动增益控制 |
| 20，21 | ANA IN、OUT | 模拟信号输入和输出 |
| 22 | *OVF | 溢出 |
| 23 | *CE | 片选（低电平允许芯片工作） |
| 24 | PD | 芯片低功耗状态控制 |
| 25 | *OEM | 录放音结束信号输出 |
| 26 | XCLK | 外部时钟 |
| 27 | P/*R | 录/放控制选择 |

## 6.3.2　语音电路

语音电路主要由 ISD2560 组成，外加扬声器、话筒等外围芯片，实现语音的录音和播放。ISD2560 的录放音控制由单片机来完成，如图 6-3 所示为具体的电路图。

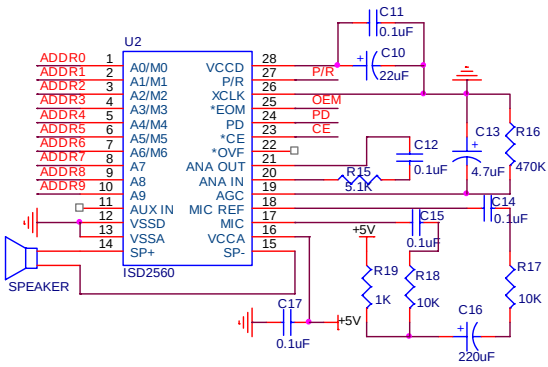



图 6-3　语音部分电路图

如图 6-3 所示，由 ISD2560 组成的语音电路比较简单。读者也可参考 ISD2560 的芯片资料中的参考设计。

### 6.3.3　单片机电路

在本系统中，单片机通过 I/O 口来控制语音芯片 ISD2560，实现如下功能：

- 按下开始键，启动录音；松开开始键，结束录音。
- 结束录音后，循环播放所录声音 2 次。

如图 6-4 所示为具体的单片机电路。

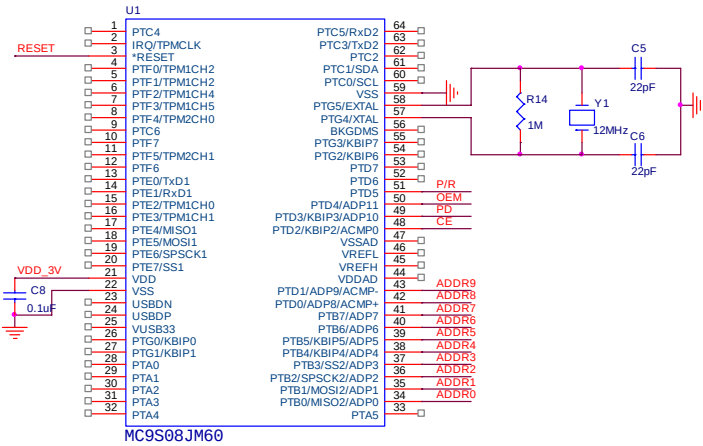



图 6-4　单片机电路图

## 6.4　软件设计与代码

### 6.4.1　软件设计流程

语音播放系统的软件设计较为简单，主要通过单片机 I/O 口对 ISD2560 的控制实现指

定地址入口的录音和循环播放。语音播放系统的主程序流程图如图 6-5 所示，单片机控制 PD、P/R 引脚为高电平，并指定录音地址，启动录音过程。在预先设定的时间内结束录音，松开“开始”键，单片机控制 P/R 引脚回到高电平，即完成一段语音的录制。之后打开中断，指定放音地址，启动放音程序，每次放音结束时，EOM 输出会触发单片机的外部中断，经过适当的延时后，重新启动第 2 次放音，这样重复 2 次后关闭外部中断，流程结束，等待下一次录音。下面给出主程序的具体程序。

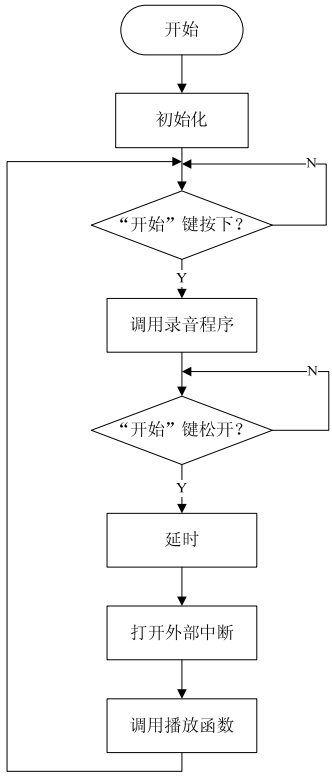



图 6-5　单片机语音播放系统的主程序流程图

## 6.4.2　初始化程序

初始化程序主要完成 ISD2560 所有端口的初始化，下面分别介绍具体的程序。

端口初始化程序主要设置相应的 I/O 管脚控制寄存器，以实现管脚的输入、输出和内部上拉等功能，键盘、LCD 和蜂鸣器等所用管脚的初始化程序也包括在内。具体程序如下：

```
void PE_low_level_init(void)
{
  /* 端口 D 上拉电阻使能寄存器设置*/
 clrSetReg8Bits(PTDPE, 0x3F, 0x40);
```

```
  /* 端口 D 数据寄存器设置 */
  clrSetReg8Bits(PTDD, 0x02, 0x2D);
  /* 端口 D 数据方向寄存器设置 */
  clrSetReg8Bits(PTDDD, 0x50, 0x2F);
  /* 端口 B 数据寄存器设置 */
  setReg8(PTBD, 0x01);
  /* 端口 B 上拉电阻使能寄存器设置*/
  setReg8(PTBPE, 0x00);
   /* 端口 B 数据方向寄存器设置 */
  setReg8(PTBDD, 0xFF);
  /* 端口 A 输出转换速率控制寄存器设置 */
  clrReg8Bits(PTASE, 0x3F);
  /* 端口 B 输出转换速率控制寄存器设置 */
  setReg8(PTBSE, 0x00);
  /* 端口 C 输出转换速率控制寄存器设置 */
  clrReg8Bits(PTCSE, 0x7F);
 /* 端口 D 输出转换速率控制寄存器设置 */
  setReg8(PTDSE, 0x00);
  /* 端口 E 输出转换速率控制寄存器设置 */
  setReg8(PTESE, 0x00);
  /* 端口 F 输出转换速率控制寄存器设置 */
  setReg8(PTFSE, 0x00);
  /* 端口 G 输出转换速率控制寄存器设置 */
  clrReg8Bits(PTGSE, 0x3F);
  /* 端口 A 输出驱动能力选择寄存器设置 */
  setReg8(PTADS, 0x3F);
  /* 端口 B 输出驱动能力选择寄存器设置 */
  setReg8(PTBDS, 0xFF);
  /* 端口 C 输出驱动能力选择寄存器设置 */
  setReg8(PTCDS, 0x7F);
  /* 端口 D 输出驱动能力选择寄存器设置 */
  setReg8(PTDDS, 0xFF);
 /* 端口 E 输出驱动能力选择寄存器设置 */
  setReg8(PTEDS, 0xFF);
 /* 端口 F 输出驱动能力选择寄存器设置 */
  setReg8(PTFDS, 0xFF);
  /* 端口 G 输出驱动能力选择寄存器设置 */
  setReg8(PTGDS, 0x3F);
/* 使能中断*/
   __EI();
}
```

### 6.4.3 ISD2560 录音、放音子程序

录音子程序负责将语音信息录入 ISD2560 芯片内，其工作过程为：在片选信号 CE 信号有效时，由 P/R 引脚选择录音/放音模式。P/R 引脚为高电平选择放音，P/R 引脚为低电平选择录音。录音时，由地址线提供起始地址，直到录音持续到 CE 或 PD 变高，或内存溢出。如果是前一种情况，芯片自动在录音结束时写入 EOM 标志。放音子程序负责将录入 ISD2560 内的语音信息播放出来，其工作过程为：在片选信号 CE 信号有效时，由 P/R 引脚选择为放音模式。放音时，由地址线提供起始地址，放音持续到 EOM 标志。如果 CE 一直为低，或芯片工作在某些操作模式，放音则会忽略 EOM 而继续进行下去，直到发生溢出为止。下面为具体的录音、放音子程序。

```
void record_voice()
{
    /*片选有效*/
    ce_PutVal(0);
    /*禁止节电控制 */
    pd_PutVal(0);
    /*芯片工作在录音模式*/
    p_r_PutVal(0);
}

void play_voice()
{
    /*片选有效*/
    ce_PutVal(0);
    /*禁止节电控制 */
    pd_PutVal(0);
    /*芯片工作在放音模式*/
    p_r_PutVal(1);
}
```

### 6.4.4 主程序

系统经过一系列初始化后，即进入主程序工作。主程序调用按键扫描程序、LCD 显示程序等来完成语音系统的各种功能，主程序中标志位的改变在按键中断服务程序和定时器中断服务程序中。下面给出语音系统的主程序。

```
void main(void)
{
   //重复播放次数计数器
    unsigned char count;
    //开始键按下标志
    unsigned char Start_Flag;
```

```
    //系统是否处于空闲状态标志
    unsigned char Idle_Flag;

PE_low_level_init();

  count = 0;
  Start_Flag = 0;
  Idle_Flag = 1;

  while(Idle_Flag == 1)
  {
    if(start_GetVal() == 0)
    {
       delay(10);
       if(start_GetVal() == 0)
       {
         Start_Flag = 1;
       }

       if(Start_Flag == 1)
       {
         do
         {
           //从地址 00 开始录音
           addr_high_PutVal(0x03);
           addr_low_PutVal(0xff);

           record_voice();
         }
         while(start_GetVal() == 1);  //开始键松开

         Start_Flag = 0;

         p_r_PutVal(1);   //结束录音
         pd_PutVal(1);    //进入节电状态

         delay(500);      //延时 500ms 再播放录音

         addr_high_PutVal(0x03); //A8,A9=0
         addr_low_PutVal(0xff);  //A0~A7 =0

         play_voice();    // 从地址 0 处进行第一次播放
```

```
            Idle_Flag = 0;   //当前不空闲，按开始键无效
          }
        }
      }
    }
```

## 6.5 实例总结

本章介绍了一种基于 MC9S08JM60 单片机的语音播放系统，采用 ISD 单片语音录放集成电路系列中的 ISD2560 实现语音的存储和播放。ISD2560 采用直接电平存储技术，这样不仅省去了 A/D、D/A 转换器，而且语音自然真实。希望读者在学习过程中要用心体会。

# 第 7 章

# 液晶显示模块：温度测量系统设计实例

单片机在测控系统中获得了广泛的应用，它除了可以测量电信号以外，还可以用于温度、湿度等非电信号的测量。能独立工作的单片机温度检测、温度控制系统已经广泛应用于很多领域。本章将从硬件、软件两个方面介绍基于 MC9S08JM60 单片机的温度测量系统。

## 7.1 实例内容说明

温度测量一般由温度传感器来进行。传统的温度检测大多以热敏电阻为温度传感器，但热敏电阻的可靠性差、测量温度准确率低，而且必须经过专门的接口电路转换成数字信号才能由单片机进行处理。而采用数字温度传感器的温度采集系统外围电路非常简单，也易于与单片机进行接口。基于这种原因，本章采用美国达拉斯（Dalls）公司生产的单线数字温度传感器来实现温度的测量，单片机采用 MC9S08JM60，外接液晶显示器，实现温度的显示。如图 7-1 所示为系统的原理框图。

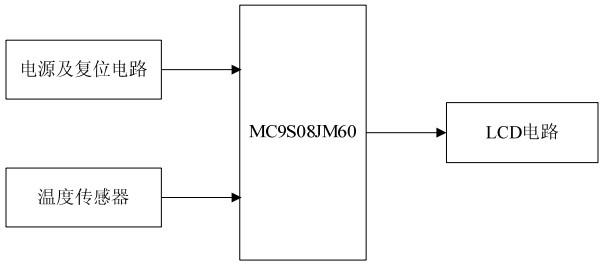



图 7-1　温度测量系统原理框图

由图 7-1 可知，温度测量系统主要由温度传感器电路、LCD 显示电路和单片机电路 3 个部分组成。温度传感器电路负责将模拟量（温度）转化成单片机可以处理的数字量，单片机将最终的温度测量结果显示在 LCD 上。

## 7.2 设计思路分析

温度测量系统的控制器选用Freescale公司基于HCS08内核系列的单片机MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、$I^2C$ 控制器、定时器等多种外设，方便进行系统设计。

分析图 7-1 所确定的系统原理框图可知，温度传感器是整个系统的关键，因此首先需选择合适的温度传感器。

**1. 温度传感器**

目前市场上的数字温度传感器芯片较多，经过综合考虑，我们选用美国达拉斯公司生产的单线数字温度传感器芯片 DS18B20 作为温度传感器。与传统的热敏电阻有所不同，DS18B20 可直接将被测温度转化成串行数字信号，以供单片机处理。

通过编程，DS18B20 可以实现 9～12 位的温度读数。信息经过单线接口送入 DS18B20 或从 DS18B20 送出，因此从微处理器到 DS18B20 仅需连接一条信号线和地线。读写和执行温度变换所需的电源可以由数据线本身提供，而不需要外部电源。

**2. 液晶显示模块**

在单片机应用系统中，通常要进行信息显示。液晶显示器（LCD）由于具有众多优点，日益成为单片机应用系统的首选显示器件。

根据显示方式和内容的不同，液晶显示模块可以分为笔段型液晶显示模块、点阵字符液晶显示模块和点阵图形液晶模块 3 种。

笔段型液晶模块是一种由段型液晶显示器件与专用的集成电路组装成一体的功能部件，只能显示数字和一些标识符号；点阵字符液晶模块是由点阵字符液晶显示器件和专用的行列驱动器、控制器及必要的连接件、结构件装配而成的，可以显示字母、数字和符号，但不能显示图形；点阵图形液晶模块的点阵像素连续排列，行和列在排布中均没有空隔，不仅可以显示字符，而且也可以显示连续、完整的图形。

显然，点阵液晶图形模块是 3 种液晶显示模块中功能最全面也最为复杂的一种，相比较，点阵字符液晶模块功能较均衡，使用也广泛。在本章的系统设计中，选用的就是 LCD1602 点阵字符液晶显示模块。

## 7.3 硬件电路设计

温度测量系统主要由温度传感器电路、LCD 显示电路、复位电路和单片机电路等组成。

本节主要介绍温度传感器电路、LCD 显示电路和单片机电路。在介绍具体的电路之前，首先对 DS18B20 进行简单的介绍。

## 7.3.1　DS18B20 介绍

DS18B20 是达拉斯公司生产的单线数字温度传感芯片，其主要功能特性如下：

- 采用单线技术，与单片机通信只需一个引脚。
- 可通过数据线供电，电压范围为 3~5.5V。
- 测量范围为-55~+125℃，在-10~+85℃范围内误差为±0.5℃。
- 用户可以从 9 位到 12 位选择数字温度计的分辨率，可配置实现 9~12 位的温度读数。
- 将 12 位的温度值转换为数字量所需时间不超过 750ms。
- 通过识别芯片各自唯一的产品序列号从而实现单线多挂接，简化了分布式温度检测的应用。
- 实际应用中不需要外部任何元器件即可实现测温。

为便于进行硬件电路的设计，下面给出该芯片的管脚图，如图 7-2 所示。

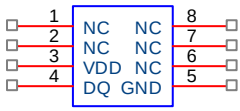



图 7-2　DS18B20 管脚图

由图 7-2 可知，该芯片只有 8 个管脚，使用起来比较简单，只需要简单的外围电路。下面对具体的管脚进行介绍：

- NC：空引脚，悬空不使用。
- VDD：可选电源管脚，电源电压范围为 3~5.5V；当工作于寄生电源时，此引脚必须接地。
- DQ：数据输入输出脚。
- GND：接地管脚。

## 7.3.2　温度传感器电路

温度传感器电路主要由 DS18B20 组成，外加一个电阻，电路非常简单，如图 7-3 所示。

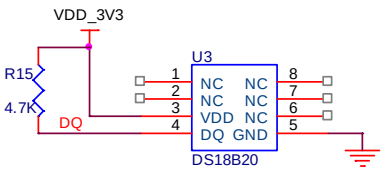



图 7-3　温度传感器电路图

如图 7-3 所示，DS18B20 和单片机 MC9S08JM60 的数据交换通过单线 DQ 来完成，DQ 引脚为漏极开路，所以需接上拉电阻。

DS18B20 有两种供电方式：寄生电源和外部电源。

寄生电源即器件从单线数据线中“窃取”电源，在信号线为高电平的时间周期内，把能力储存在内部的电

容器中，在信号线为低电平的时间周期内断开此电源，直到信号线变为高电平，重新接上寄生（电容）电源为止。寄生电源的优点为：

- 可实现远程温度检测而无需本地电源；
- 没有正常电源的条件下也可以读 ROM。

为了使 DS18B20 能完成准确的温度转换，当温度变换发生时，DQ 线上必须提供足够的功率。4.7kΩ 的上拉电阻将使得 DQ 线没有足够的驱动能力。如果有多个 DS18B20 挂接，而且同时变换时，这一问题将变得更为突出。解决的方法是在发生温度变换时，在 DQ 线上提供强的上拉。

DS18B20 的另一种供电方式是将 VDD 引脚接外部电源（3~5.5V）。这种接法的优点是在 DQ 线上不要求强的上拉。总线上的主机在温度变换期间不需要一直使 DQ 线保持高电平，这就允许在变换期间内其他数据在单线上传送。而且单线上可以放置多个 DS18B20。如果它们都是用外部电源，那么通过发起“跳过 ROM”命令，接着执行“温度变换”命令就可以同时完成各自的温度变换。

本例中只用了一片 DS18B20，由于不存在远程温度测量的考虑，所以为了简单起见，仍然采用外部供电的方式。

### 7.3.3 液晶显示电路

LCD 电路用于显示测量的温度，由于 MC9S08JM60 本身不带 LCD 控制器，因此我们选用内置 LCD 驱动器的 LCD 模块，如图 7-4 所示为 LCD 显示电路图。

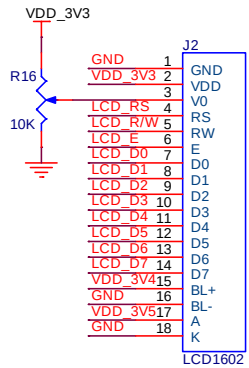



图 7-4 液晶显示电路图

从图 7-4 可以看出，A、K 引脚用于背光，本例中为简化处理，直接接电源和地。数据线 D0～D7 和单片机的 PTB 口相连。RS 为寄存器选择，高电平时选择数据寄存器，低电平时选择指令寄存器。RW 为读写信号线，高电平时进行读操作，低电平时进行写操作，当 RS 和 RW 共同为低电平时可以写入指令或者显示地址，当 RS 为低电平并且 RW 为高电平时可以读忙信号，当 RS 为高电平并且 RW 为低电平时可以写入数据。E 端为使能端，当 E 端由高电平跳变成低电平时，液晶模块执行命令。控制线 RS、R/W 和使能线分别与单片机 PTC0～PTC2 引脚相连。

### 7.3.4 单片机电路

在本系统中，单片机通过单线 DQ 从 DS18B20 读出测量到的温度数据。由于 DS18B20 支持单线技术，MC9S08JM60 单片机利用一个 I/O 口来模拟单线协议，实现与 DS18B20 接口。此外，单片机还通过模拟 I/O 方式外接了一个 LCD 模块，实现测量温度的显示。如图 7-5 所示为具体的单片机电路。

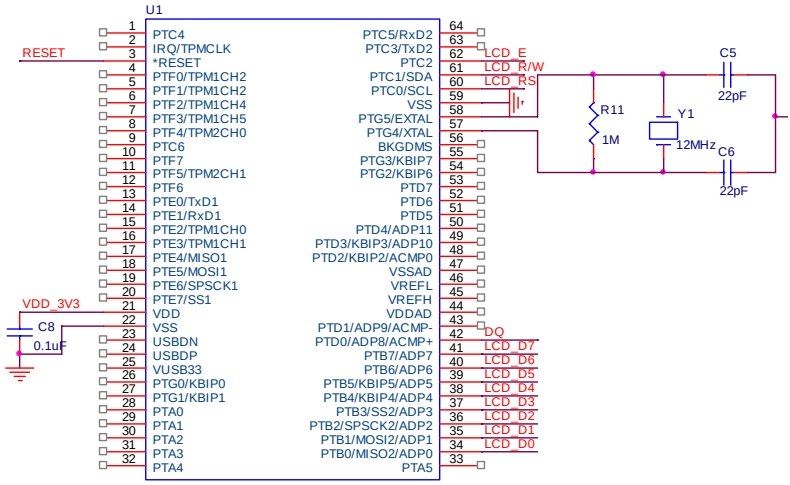



图 7-5　单片机电路图

# 7.4　软件设计与代码

## 7.4.1　软件设计流程

温度测量系统的软件主要包括温度测量程序、LCD 显示程序和主程序 3 个部分。温度测量程序主要通过单片机 I/O 来模拟单线协议实现和 DS18B20 的接口，从而读取测量到的温度值。LCD 显示程序实现将测量的温度值显示在 LCD 上。温度测量系统的主程序流程图如图 7-6 所示。单片机初始化后，进入主循环，读取 DS18B20 测量的温度值，然后送到 LCD 显示。下面给出主程序的具体程序。

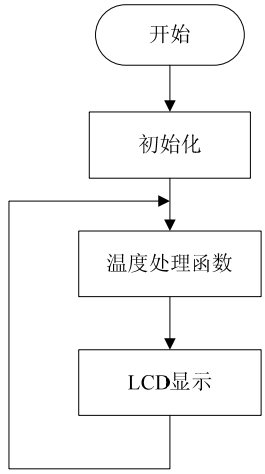



图 7-6　温度测量系统的主程序流程图

## 7.4.2　温度测量程序

在本系统中，单片机读取温度传感器的数据是通过单线（1-wire）技术来实现的，因此在介绍具体程序之前先介绍单线技术。

**1. DS18B20的单线协议**

DS18B20 有严格的通信协议来保证各位数据传输的正确性和完整性。

主机操作单线器件 DS18B20 必须遵循下面的顺序：

（1）初始化

单线总线上的所有操作均从初始化开始。初始化过程如下：主机通过拉低单线 480us 以上，产生复位脉冲。然后释放该线，进入 Rx 接收模

式。主机释放总线时，会产生一个上升沿。单线器件 DS18B20 检测到该上升沿后，延时15～60us ，通过拉低总线 60～240us 来产生应答脉冲。主机接收到从机的应答脉冲后，说明有单线器件在线。

（2）ROM 操作命令

一旦总线主机检测到应答脉冲，便可以发起 ROM 操作命令。

（3）内存操作命令

在成功执行了 ROM 操作命令后，才可以使用内存操作命令。

（4）数据处理

DS18B20 要求有严格的时序来保证数据的完整性。在单线 DQ 上，存在复位脉冲、应答脉冲、写“0”、写“1”、读“0”和读“1”几种信号类型。其中，除了应答脉冲之外，均由主机产生。复位和应答脉冲在前文中已经介绍了，而数据位的读和写则是通过读、写时隙实现的。

所有的读写时隙至少需要 60us，且每两个独立的时隙之间至少需要 1us 的恢复时间。在写时隙中，主机在拉低总线 15us 内释放总线，并向 DS18B20 写“1”。若主机拉低总线后能保持至少 60us 的低电平，则向单总线器件写“0”。DS18B20 仅在主机发出读时隙时才向主机传输数据，所以，当主机向 DS18B20 发出读数据命令后，必须马上产生读时隙，以便 DS18B20 能传输数据。

**2. 温度测量程序的实现**

温度测量程序主要是完成温度的读出操作，温度的读取主要是通过操作 DS18B20 的相应寄存器来实现的。关于 DS18B20 的各个寄存器在此不作介绍，读者可以参看 DS18B20 的数据手册。下面为具体的程序：

```
unsigned char get_temp(void)          //外接 5V 电源情况
{
  int i;

  DQ_SetDir(0);
  reset();                           //复位等待从机应答
  write_byte(0XCC);                   //忽略 ROM 匹配
  write_byte(0X44);                   //发送温度转化命令

  for(i=50;i>0;i--)
  {
    feedCOP();                         //确保温度转换完成所需要的时间
  }

  reset();                           //再次复位，等待从机应答
  write_byte(0XCC);                   //忽略 ROM 匹配
  write_byte(0XBE);                   //发送读温度命令
```

```
    TLV=read_byte();                          //读出温度低 8 位
    THV=read_byte();                          //读出温度高 8 位
    DQ_SetVal();                               //释放总线
    TZ=(TLV>>4)|(THV<<4)&0X3f;                 //温度整数部分

    asm
    {
      NOP
      NOP
      NOP
      NOP
    }
    return TZ;
}
```

### 7.4.3　LCD 显示程序

本系统中，选用内置 HD44780 驱动器的 1602 LCD 模块。HD44780 控制器的数据手册可去网上下载。对 LCD1602 的读写操作、屏幕和光标的操作都是通过对 HD44780 内部寄存器编程来实现的。下面给出 LCD 显示的子程序。

```
static  void  Lcd_Disp_CursorSet (Uchar row, Uchar col)
{
   switch (row) {
      case 0:
          /* LCD 仅有一行*/
          if (DispMaxRows == 1)
          {
              if (col < (DispMaxCols >> 1))
              {
                 /* 第一行的前半列起始地址为 0x80*/
                 Lcd_Write_Data(0x80 + col);
              }
              else
              {
                 /* 第一行的后半列起始地址为 0xC0   */
                 Lcd_Write_Data(0xC0 + col - (DispMaxCols >> 1));
              }
          }
          else
          {
             Lcd_Write_Data(0x80 + col);
            }
          break;
```

```
        case 1:
             /* 选择 LCD 显示的第 2 行*/
             Lcd_Write_Data(0xC0+col);
            break;

        case 2:
            /* 选择 LCD 显示的第 3 行*/
             Lcd_Write_Data(0x80 + DispMaxCols + col);
             break;

        case 3:
             /* 选择 LCD 显示的第 4 行*/
             Lcd_Write_Data(0xC0 + DispMaxCols + col);
    }
}

void  Lcd_Disp_Str (Uchar row, Uchar col, char *s)
{
    unsigned char i;

    if (row < DispMaxRows && col < DispMaxCols)
    {
        /* 定位光标在指定的行、列*/

        Lcd_Disp_CursorSet(row, col);
        i = col;
         while (i < DispMaxCols && *s) {
             /* 显示一列字符 */
             Lcd_Write_Data(*s++);
             /*递增计数器 */
             i++;
        }
    }
}
```

### 7.4.4 主程序

系统经过一系列初始化后，即进入主程序工作。主程序调用温度测量程序、LCD 显示程序等来完成温度测量系统的功能。下面给出温度测量系统的主程序。

```
void main(void)
{
     PE_low_level_init();
     Lcd_Init();                //LCD 初始化
```

```
        Lcd_Disp_Str(0,1,LCDBuf);

        for(;;){
       get_temp();                  //读取 DS18B20 的温度值
       WaitNms(100);
       Lcd_Disp_Str(1,5,&TZ);      //将温度值显示在 LCD 上
       WaitNms(1000);
     }
   }
```

## 7.5　实例总结

本章介绍了一种基于 MC9S08JM60 单片机实现的温度测量系统。系统的硬件设计采用 DS18B20 芯片来实现。在软件设计时主要介绍了单线（1-wire）技术和 LCD 显示程序的实现。本章的实例虽然相对简单，但实现的温度测量系统具有很强的代表性，可以很容易地移植到其他单片机平台上，具有很强的应用价值。

# 第 8 章

# 液晶显示模块：实时时钟设计实例

在单片机应用系统中，常常需要记录实时的时间信息。比如在采集系统中，对某些重要的信息不仅需要记录其内容，还需要记录下该事件发生的准确时间，由此可见，实现一个实时时钟是非常有必要的。本章将详细介绍一个基于 MC9S08JM60 单片机实现的实时时钟系统。

## 8.1 实例内容说明

实时时钟（RTC）可应用于许多领域：从钟表到时间标记事件，甚至到产生事件。在电力自动化、工业控制等自动化程度较高的领域，大多数设备都处于无人值守的状态，所以要求在发生故障时将故障时间和相关信息记录下来，以便进行具体分析。目前，虽然市面上有很多专用的 RTC 器件，但这些器件系统集成度低，因此本章采用 MC9S08JM60 单片机实现通用的实时时钟系统。

由于 MC9S08JM60 单片机带有实时计数器模块，因此基于 MC9S08JM60 单片机的实时时钟系统结构非常简单，仅需单片机最小系统，外接一个 LCD 显示模块即可。如图 8-1 所示为系统的原理框图。

由图 8-1 可知，实时时钟系统结构比较简单，由按键和 LED 电路、LCD 显示电路和单片机电路 3 部分组成。按键和 LED 电路主要用于工作模式的切换。单片机电路负责参数实时时钟信息，如时、分、秒、年、月、日等，LCD 显示电路负责显示实时时间信息。

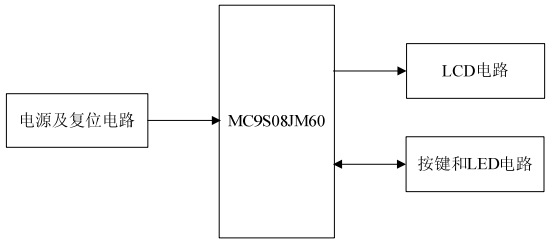



图 8-1　系统原理框图

## 8.2　设计思路分析

由于系统要实现的功能比较单一（主要就是获取实时时间信息），并将时间显示在 LCD 上。因此设计思路非常清晰，关键在于实时时钟的实现。

实时时钟的实现主要有硬件和软件两种方式。硬件方式即使用外部 RTC 硬件模块（常集成 IIC 接口）或有的单片机内部集成了一个 RTC 模块。软件方式主要是利用定时器或计数器来产生指定间隔的时间中断。记录间隔的次数，并将之转化为时间。通常采用 1s 的定时间隔中断。

硬件方式的实时时钟多用在对时间精度要求比较严格的场合；软件方式的实时时钟无需外围器件，硬件成本低，但软件设计较硬件方式要复杂。本例我们选用硬件方式，即 MC9S08JM60 单片机内部集成的一个 RTC 模块。

## 8.3　硬件电路设计

实时时钟系统主要由按键和 LED 电路、LCD 显示电路、复位电路和单片机电路等组成。电源电路、复位电路和 BDM 仿真调试电路请参考第 5 章。本节主要介绍按键和 LED 电路、LCD 显示电路和单片机电路。

### 8.3.1　按键和 LED 电路

按键和 LED 电路用于实时时钟实例不同工作模式的切换和状态的显示。本例实现了 4 个独立按键和 5 个 LED 发光二极管。如图 8-2 所示为按键和 LED 电路。

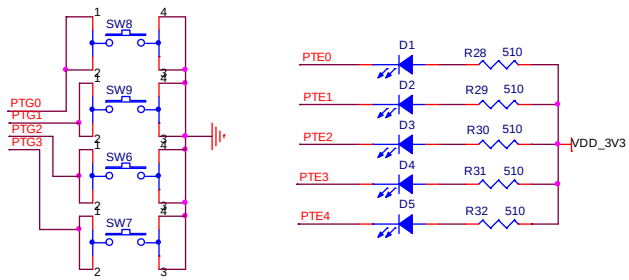



图 8-2　按键和 LED 电路图

### 8.3.2 液晶显示电路

LCD 电路用于显示时间，由于 MC9S08JM60 本身不带 LCD 控制器，因此我们选用内置 LCD 驱动器的 LCD 模块。如图 8-3 所示为 LCD 显示电路图。

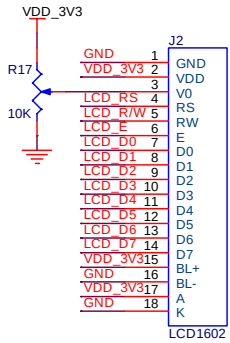



图 8-3 液晶显示电路图

从图 8-3 可以看出，A、K 引脚用于背光，本例中为简化处理，直接接电源和地。数据线 D0～D7 和单片机的 PTB 口相连。RS 为寄存器选择，高电平时选择数据寄存器，低电平时选择指令寄存器。RW 为读写信号线，高电平时进行读操作，低电平时进行写操作。当 RS 和 RW 共同为低电平时可以写入指令或者显示地址；当 RS 为低电平 RW 为高电平时可以读忙信号；当 RS 为高电平 RW 为低电平时可以写入数据。E 端为使能端，当 E 端由高电平跳变成低电平时，液晶模块执行命令。控制线 RS、R/W 和使能线分别与单片机 PTC0～PTC2 引脚相连。

### 8.3.3 单片机电路

在本系统中，单片机通过内部 RTC 计数器获取实时时间信息。单片机通过模拟 I/O 方式外接了一个 LCD 模块，将获取的实时时间显示出来。如图 8-4 所示为具体的单片机电路。

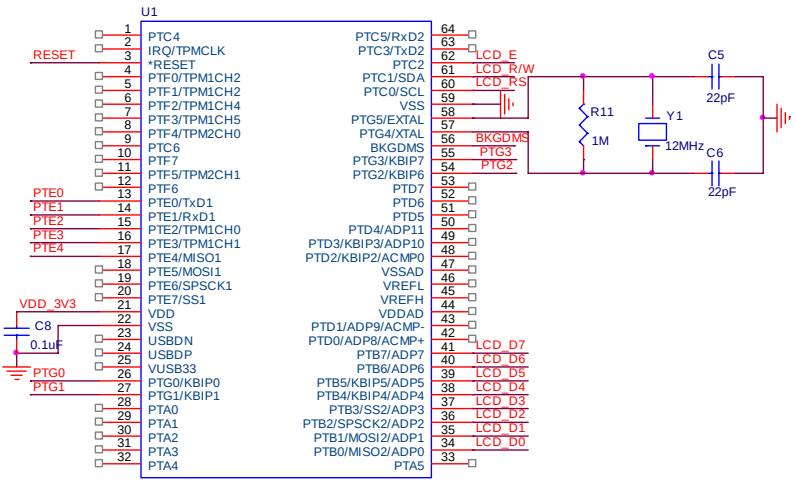



图 8-4 单片机电路图

## 8.4 软件设计与代码

### 8.4.1 软件设计流程

实时时钟系统的软件主要包括时钟子程序、人机接口程序和主程序 3 个部分。其中时

钟子程序是系统的核心，负责完成实时时钟日期和时间的设置和读取。人机接口程序包括 LCD 显示程序、LED 控制程序和独立按键程序等。LCD 显示程序实现将实时时钟信息显示在 LCD 上。实时时钟系统的主程序流程图如图 8-5 所示。单片机初始化后，进入主循环，不停地判断是否有按键按下，如果有按键按下，则执行相应的功能。

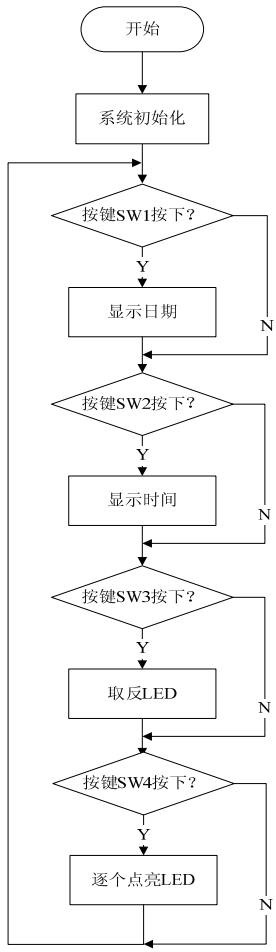



图 8-5　实时时钟主程序流程图

## 8.4.2　时钟子程序

时钟子程序主要是完成实时时钟日期、时间的设置/读取。下面给出日期/时间设置功能的具体程序。

```
byte TimeDate_SetDate(word Year,byte Month,byte Day)
{
  byte a,b,c,d,tDay,tW;
  const byte * ptr;                    //指向 ULY/LY 表的指针变量
```

```
  if ((Year < 1998) || (Year > 2099) || (Month > 12) || (Month == 0) ||
        (Day > 31) || (Day == 0))  //检测日期参数是否合理
{
    return ERR_RANGE;
  }

  ptr = ( Year & 0x03 ? ULY : LY );    //指针指向闰年或非闰年定义表

  if (ptr[Month-1] < Day)              //检测天数是否超出范围
  {
    return ERR_RANGE;
  }
  a = (byte)( Year > 1999 ? 6 : 0 );   //计算属于一周的哪一天
  b = (byte)(Year % 100);
  c = (byte)(b >> 2);
  tDay = Day;
  if (!(Year & 3) && (Month < 3)) {
    tDay--;
  }
  d = Month_Table[Month];
  tW = (byte)((a + b + c + d + tDay) % 7);

  EnterCritical();                     //关中断
  CntDOW = tW;
  CntDay = Day;
  CntMonth = Month;
  CntYear = Year;
  ExitCritical();                     //开中断
  return ERR_OK;
}

byte TimeDate_SetTime(byte Hour,byte Min,byte Sec,byte Sec100)
{
  bool TmrRun;
  if ((Sec100 > 99) || (Sec > 59) || (Min > 59) || (Hour > 23))
  { /* 判断时间参数是否合理? */
    return ERR_RANGE;                 //如果不合理，则置错误标志位
  }
  TmrRun = EnUser;
  if (EnUser) {                       //RTC 模块使能标志被置 1?
    EnUser = FALSE;
    HWEnDi();                         //根据状态标志使能设备
  }
```

```
  /* 将需设置的时间参数转化成单位为 10ms 节拍的值*/
  TotalHthH = (360000 * (dword)Hour) + (6000 * (dword)Min) + (100 *
(dword)Sec) + (dword)Sec100;
  TotalHthL = 0;
  AlarmFlg = (bool)!(TotalHthH < AlarmHth); //设置报警标志
  if (TmrRun) {
    EnUser = TRUE;
    HWEnDi();
  }
  return ERR_OK;
}
```

### 8.4.3　LCD 显示程序

本系统中，选用内置 HD44780 驱动器的 1602 LCD 模块。HD44780 控制器的数据手册可去网上下载。对 LCD1602 的读写操作、屏幕和光标的操作都是通过对 HD44780 内部寄存器编程来实现的。下面给出 LCD 显示的子程序。

```
void Lcd_Show(unsigned char str[])
{
    unsigned char i;

    /*调用 Lcd 初始化子程序*/
    Lcd_Init();

    /*显示第 1 行*/
    /*RS、R/W=00,写指令*/
    Lcd_RS_PutVal(0);
    Lcd_RW_PutVal(0);
    Lcd_Command(0b10000000);

    /*RS、R/W=10,写数据到 DDRAM 中*/
    Lcd_RS_PutVal(1);
    Lcd_RW_PutVal(0);
    /*逐个显示*/
     for (i=0;i<10;i++)
     {
       Lcd_Command(str[i]);
     }

     /*显示第 2 行*/
     /*RS、R/W=00,写指令*/
     Lcd_RS_PutVal(0);
     Lcd_RW_PutVal(0);
```

```
    Lcd_Command(0b11000000);

    Lcd_RS_PutVal(1);
   Lcd_RW_PutVal(0);
   for (i=10;i<18;i++)
   {
     Lcd_Command(str[i]);
   }
}
```

### 8.4.4 主程序

系统经过一系列初始化后，即进入主程序工作，主程序不断查询是否有按键按下，有按键按下则执行相应的功能。下面给出实时时钟的主程序。

```
void main(void)
{
    byte  a[] = "Date:";
    byte  b[] = "Time:";
    byte LEDarray[5] = {FALSE, FALSE,FALSE,FALSE,FALSE}, LEDcounter, i,j;
    byte c;
    TIMEREC Time;
    DATEREC Date;

    PE_low_level_init();

    SW4toggle = FALSE;
   counter = 0;
   LEDcounter = 0;
   pastminute = 0;
   pastday = 0;

    for(;;)
    {
     //按键 1 按下，则显示日期
     result = TimeDate_GetDate( &Date );
     NewSW1 = SW1_GetVal();// >> 4;
     if (  (Date.Day != pastday) || (NewSW1 != OldSW1)  )
     {
       Lcd_Show_Line1(a, 6);

       if (NewSW1 == 0)
       {
           Lcd_Show_Line2(Date.Month);
           Lcd_Show_Line2(':');
```

```
        Lcd_Show_Line2(Date.Day);
        Lcd_Show_Line2(':');
        Lcd_Show_Line2(Date.Year);
    }
    else if   (Date.Day != pastday)
    {
        Lcd_Show_Line2(Date.Month);
        Lcd_Show_Line2(':');
        Lcd_Show_Line2(Date.Day);
        Lcd_Show_Line2(':');
        Lcd_Show_Line2(Date.Year);
    }
    pastday =  Date.Day;
    OldSW1 = NewSW1;
  }

  //按键 2 按下，则显示时间
result = TimeDate_GetTime( &Time );
  NewSW2 = SW2_GetVal();
  if ((Time.Min != pastminute) || (NewSW2 != OldSW2) )
  {
     Lcd_Show_Line1(b, 6);

        if ((NewSW2 == 0)) //是否有按键按下?
        {
        if (Time.Hour > 12)
        {
            Lcd_Show_Line2(Time.Hour-12);
            Lcd_Show_Line2(':');
            Lcd_Show_Line2(Time.Min);
            Lcd_Show_Line2(':');
            Lcd_Show_Line2(Time.Sec);
        }
        else
        {
            Lcd_Show_Line2(Time.Hour);
            Lcd_Show_Line2(':');
            Lcd_Show_Line2(Time.Min);
            Lcd_Show_Line2(':');
            Lcd_Show_Line2(Time.Sec);
        }
        }
        else if (Time.Min != pastminute)      //每分钟更新一次
        {
```

```
        if (Time.Hour > 12)
        {
            Lcd_Show_Line2(Time.Hour-12);
            Lcd_Show_Line2(':');
            Lcd_Show_Line2(Time.Min);
            Lcd_Show_Line2(':');
            Lcd_Show_Line2(Time.Sec);
        }
        else
        {
            Lcd_Show_Line2(Time.Hour);
            Lcd_Show_Line2(':');
            Lcd_Show_Line2(Time.Min);
            Lcd_Show_Line2(':');
            Lcd_Show_Line2(Time.Sec);
        }
        }
        pastminute =  Time.Min;
        OldSW2 = NewSW2;
    }

    //按键 SW3 按下时执行的功能
      NewSW3  = SW3_GetVal();// >> 6;
      if ( (NewSW3 != OldSW3) )
      {
          if (  NewSW3 == 0 )
          {
              LED_PutVal(~LED_GetVal());
          }
          OldSW3 = NewSW3;
      }

      //按键 SW4 按下时的处理程序
      NewSW4 = SW4_GetVal();// >>7;
      counter = counter + 1;

      if (NewSW4 == 0)
      {
        SW4toggle = ~SW4toggle & 1;
      }
     if (  (NewSW4 != OldSW4) || (SW4toggle == TRUE) )
     {
          if (  counter >= 3000 )
          {
```

```
            for (i=0; i <= 5; i++)  LEDarray[i] = FALSE;
            LEDarray[LEDcounter] = TRUE;
            LED_PutBit(0, LEDarray[0]);
            LED_PutBit(1, LEDarray[1]);
            LED_PutBit(2, LEDarray[2]);
            LED_PutBit(3, LEDarray[3]);
            LED_PutBit(4, LEDarray[4]);
               LEDcounter =  LEDcounter + 1;
               if (LEDcounter >= 6) LEDcounter = 0;
            counter = 0;
          }

          OldSW4 = NewSW4;
      }
  }
```

## 8.5　实例总结

本章介绍了如何实现基于 MC9S08JM60 单片机的实时时钟设计。主要利用 MC9S08JM60 单片机内部的实时计数模块，辅之以 LCD 显示模块，设计出方便、运行可靠的实时时钟系统。本实例设计过程虽然简单，但其设计思想也可以在其他基于单片机的实时时钟系统中利用，读者学完后应加强融会贯通。

# 第 9 章

# KBI 模块：扫描键盘设计实例

在单片机系统中，人机接口是非常重要的组成部分，如需要输入配置信息参数等。在这些应用中，键盘是一个必不可少的部分。键盘设计合理，不仅可以节省系统的设计成本，还可使仪器设备的操作变得更为简单、方便，在很大程度上提高系统的综合性能。本章将介绍一个基于 MC9S08JM60 单片机实现的矩阵键盘，首先介绍该系统的硬件设计，然后再介绍其软件设计。

## 9.1 实例内容说明

矩阵式键盘，即通常所讲的行列式键盘，由行线和列线组成，按键位于行、列的交叉点上，行、列分别连接到按键开关的两端，行线通过上拉电阻接到高电平。无按键动作时，行线处于高电平状态；有按键按下时，交点的行线和列线接通，行线电平状态将由与此行线相连的列线电平决定。列线电平如果低，则行线电平为低；列线电平如果高，则行线电平也为高。这一点是识别矩阵键盘按键是否被按下的关键所在。由于矩阵键盘中行、列线为多键公用。各按键均影响该键所在的行和列的电平，所以必须将行、列线信号配合起来作适当的处理，才能确定闭合键所在的位置。矩阵式键盘节省了很多的 I/O 口，适用于按键数量比较多的场合。

在应用中，如果直接采用循环扫描方式，则降低了微控制器的使用效率，所以在一般情况下，大都采用键盘中断方式，利用内部定时器，产生一定时间的定时器中断，并设置按键标志。当按键标志位有效时才对键盘进行扫描。MC9S08JM60 有 8 个键盘中断引脚，可利用其键盘中断方式实现微控制器的最大工作效率。如图 9-1 所示为键盘扫描设计系统的原理框图，通过按键来改变 LED 数码管相应的显示内容。

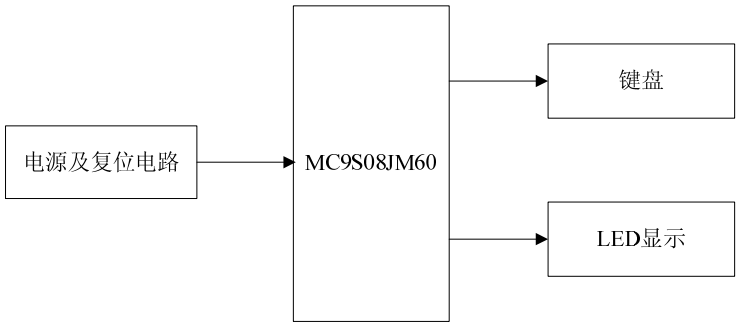



图 9-1　系统原理框图

由图 9-1 可以看出，系统的硬件电路比较简单，下面对具体的设计思路进行介绍。

## 9.2　设计思路分析

要实现矩阵式键盘的按键识别，首先必须了解矩阵式键盘的工作原理。下面以 4×4 矩阵式键盘为例，说明矩阵键盘扫描的基本原理。

如图 9-2 所示，列线（C1～C4）通过电阻接+3.3V，当键盘上没有键被按下时，所有的行线和列线短路。例如，第 2 行第 3 列的按键被按下闭合时，行线 L2 和列线 C3 短路，此时 C3 线上的电平由 L2 的电位所决定。那么如何确定键盘上哪个键被按下了呢?可以把列线 C1～C4 接到微控制器的输入口，行线 L1～L4 接到微控制器的输出口。在微控制器的控制下，使行线 L1 为低电平（0），其余 3 根行线 L2～L4 都为高电平，读列线 C1～C4 状态；如果读出列线 C1～C4 的状态都为高电平，则 L1 这行没有键按下，如果读出列线 C1～C4 的状态不全为高电平，那么为低电平列线和 L1 相交的键被按下；如果 L1 这一行没有键被按下，接着使行线 L2 为低电平，其余行线为高电平，用同样的方法检查 L2 这一行有无键按下；以此类推，最后使行线 L4 为低电平，其余的行线为高电平，检查 L4 这一行上是否有键被按下。这种逐行逐列检查键盘状态的过程称为对键盘的行列式扫描。

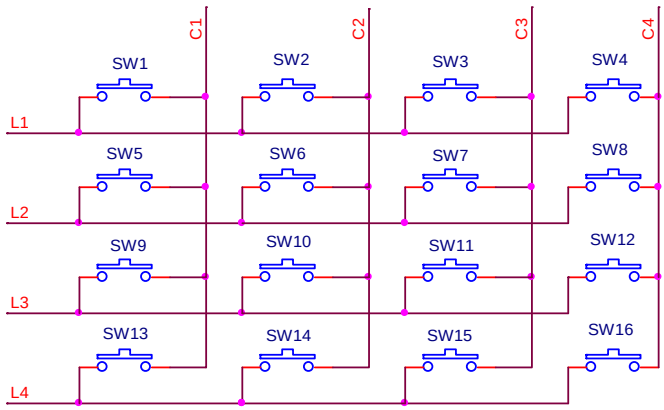



图 9-2　4×4 矩阵式键盘

CPU 对键盘扫描可以采取程控的随机方式，在 CPU 空闲时扫描键盘；也可以采取定时控制，每隔一定时间，CPU 对键盘扫描一次。CPU 可随时响应键输入请求，也可以采用中断方式，当键盘有键被按下时，向 CPU 请求中断，CPU 响应键盘输入中断，对键盘扫描，识别哪一键被按下，并对键输入信息作出相应处理。CPU 对键盘上被按键的键号确定，可以根据行线和列线的状态计算求得，也可以根据行线和列线的状态查表求得。

## 9.3 硬件电路设计

键盘扫描设计实例主要由电源电路、LED 显示电路、按键电路和单片机电路组成。在本系统中，要求的电源电压为 3.3V，由于 3.3V 电源电路在第 5 章已经作了介绍，因此本章不再介绍。下面分别介绍按键电路、LED 显示电路和单片机电路。

### 9.3.1 按键电路

4×4 矩阵式键盘的电路如图 9-3 所示。MC9S08JM60 的 PTB0、PTB1、PTB2 和 PTB3 作为矩阵键盘的行控制线，配置为输出；PTG0、PTG1、PTG2 和 PTG3 作为矩阵键盘的列线输入，配置为输入，内部上拉使能，这样在没有按键按下时列线为高电平输入，当有按键按下时，对应的 I/O 口被拉低，这样检测 I/O 口电平状态即可判断按键按下与否。

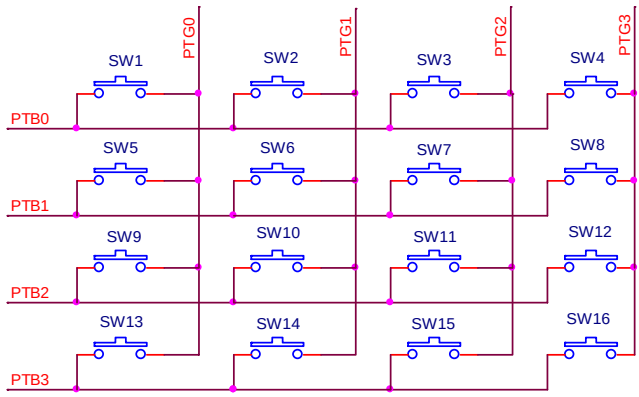



图 9-3 按键电路图

### 9.3.2 LED 显示电路

LED 数码管电路用于按键按下时的状态变换显示，采用 4 位 LED 数码管，动态显示的方法。E 口的 8 位对数码管进行段选，B 口的高 4 位通过驱动对数码管进行位选。如图 9-4 所示为 LED 数码管动态显示电路图。

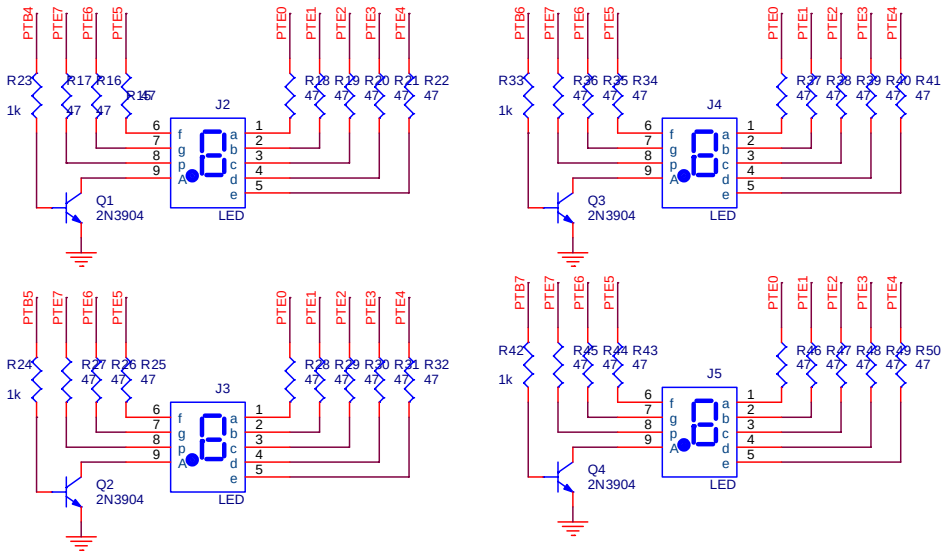



图 9-4　LED 动态显示电路图

## 9.3.3　单片机电路

单片机的电路比较简单，只需要单片机的 PTB0、PTG0～PTG3、PTE0 与键盘电路和 LED 数码管电路接口。另外单片机还需相应的振荡电路才能使单片机进行工作。如图 9-5 所示为具体的单片机电路图。

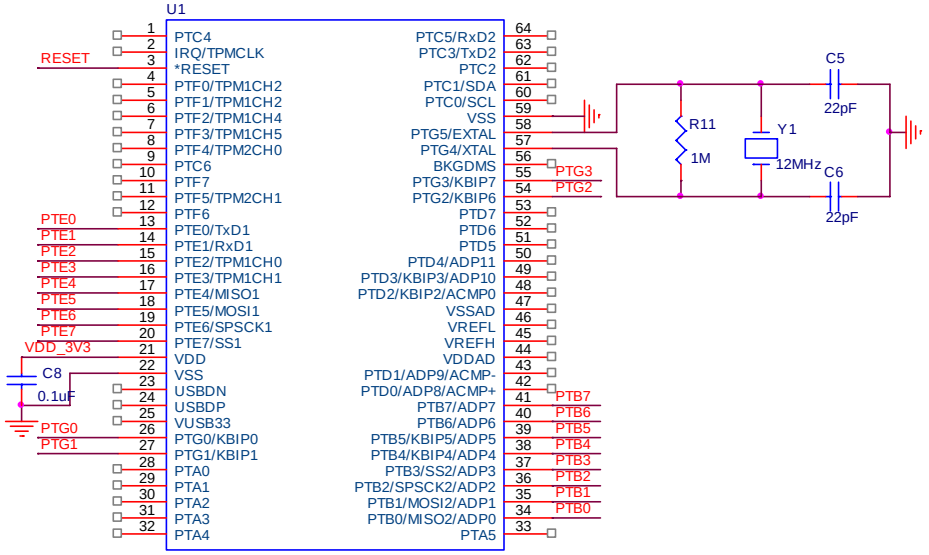



图 9-5　单片机电路图

# 9.4 软件设计与代码

## 9.4.1 软件设计流程

对于一般的 I/O 端口来说，采用查询方式实现软件，但是 MC9S08JM60 单片机集成了键盘中断模块（KBI），因此可以采用中断方式实现软件。系统主要包括键盘扫描子程序、LED 数码管显示子程序和主程序 3 个部分。下面分别介绍具体的程序。

## 9.4.2 键盘扫描子程序

本例使用键盘中断方式编程。在中断方式中，行线初始化便被置于低电平，因此，当有键被按下时，便会引起 PTG0～PTG3 中的相应位由高电平变为低电平，产生一个下降沿，从而触发键盘中断。在键盘服务子程序中，才会进行键盘扫描，判断键值。下面介绍具体的程序。

```
void keyscan(void)
 {
   unsigned char data,k0;

   for(data=0;data<4;data++)
   {
      PTBD=0x0f;
      switch(data)
      {
        case 0:
          PTBD=0x0e;         //设置 ptb0=0
          break;

        case 1:
          PTBD=0x0d;
          break;            //设置 ptb1=0

        case 2:
          PTBD=0x0b;
          break;            //设置 ptb2=0

        case 3:
          PTBD=0x07;
          break;            //设置 ptb3=0
      }

      k0=PTGD&0x0f;
```

```
      if(k0!=0x0f)
      {
        if(k0==0x0e)        //第 1 列有键被按下，键值为 data
        {
          k0=data;
        }
        if(k0==0x0d)        //第 2 列有键被按下，键值为 data+4
        {
          k0=4+data;
        }
        if(k0==0x0b)        //第 3 列有键被按下，键值为 data+8
        {
          k0=8+data;
        }
        if(k0==0x07)        //第 4 列有键被按下，键值为 data+12
        {
          k0=12+data;
        }

        key_number=k0;       //将键值存入 key_number 中
      }
    }
  }

void kbi_OnInterrupt(void)
{
   setReg8(KBIPE, 0x00);
   clrReg8Bits(KBISC, 0x02); //禁止键盘产生硬件中断

   keyscan();              //调用键盘扫描子程序
   key_line_PutVal(0x00);
   setReg8Bits(KBISC, 0x02);//允许键盘产生中断

   setReg8(KBIPE, 0xC3);     //定义 PTG0~PTG3 引脚为中断输入引脚
}
```

### 9.4.3　LED 显示程序

LED 数码管显示程序将按键被按下时相应的数值显示出来。由于数码管位数较多，因此我们采取动态显示的方法。下面给出 LED 动态显示的子程序。

```
void display(unsigned char num,unsigned char n)
{
  unsigned char i,j,k=8;
```

```
  led_bits_PutVal(0x00);     //关显示

  for(j=0;j<(n+1);j++)
  {
    k*=2;
  }

  led_section_PutVal(ledsccode[num]);  //将编码送入 E 口
  led_bits_PutVal(k);        //位选

  for(i=0;i<255;i++)         //数码管发亮延时，使字符能够清晰显示
  {
    asm(nop);
    asm(nop);
  }
}
```

### 9.4.4 主程序

系统经过一系列初始化后，即进入主程序工作。主程序调用按键扫描程序、LED 显示程序等来完成键盘扫描设计系统，键盘扫描采用中断方式编程。下面给出扫描键盘设计实例的主程序。

```
void main(void)
{
    unsigned char h,l;
    PE_low_level_init();

    for(;;)
    {
      if(key_number==1)
      {
        for(i=0;i<9;i++)
        {
          for(j=1;j<255;j++)
          {
            display(key_number,0);
            display(i,3);
          }
        }
      }
      else if(key_number==2)
      {
        for(i=9;i>0;i--)
```

```
            {
              for(j=1;j<255;j++)
              {
                display(key_number,0);
                display(i,3);
              }
            }
          }
          else
          {
            display(key_number,0);
            display(15,2);
          }
        }
      }
```

## 9.5　实例总结

本章通过利用单片机内部的 KBI 模块，辅之以简单的外围电路，设计出了矩阵式键盘扫描显示系统。显示部分采用 4 位 LED 数码管，使用动态显示方式。单片机采用的是 MC9S08JM60，由于其支持键盘中断，接口丰富，因此简化了程序的设计。读者学习后，可以根据 MC9S08JM60 集成模块众多、接口丰富的特点，进行更复杂的系统构建设计。

# 第 10 章

# 定时器模块：用定时器控制蜂鸣器发声

蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于家用电器、计算机、打印机、复印机、报警器、电子玩具、汽车电子设备、电话机、定时器等电子产品中作为发声器件。本章将介绍一个基于 MC9S08JM60 单片机的定时器模块实现对蜂鸣器发声控制的实例。首先介绍该系统的硬件设计，然后再介绍其软件设计。

## 10.1 实例内容说明

蜂鸣器作为一种常用的发声器件，根据其内部是否具有振荡器，分为有源蜂鸣器和无源蜂鸣器；根据蜂鸣器所采用的材料，主要分为压电式蜂鸣器和电磁式蜂鸣器。本文实现的是用 MC9S08JM60 单片机的定时器模块来控制无源蜂鸣器的发声，其系统的原理框图如图 10-1 所示。

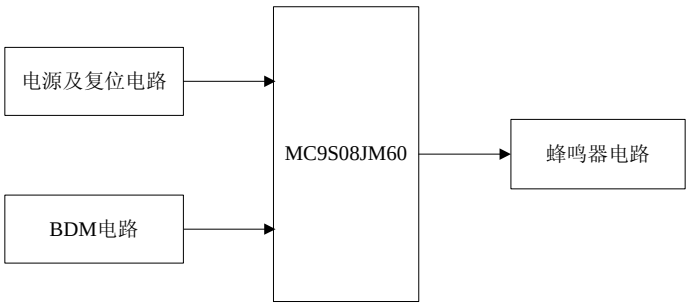



图 10-1　系统原理框图

由图 10-1 可以看出，系统的硬件电路比较简单，主要由两部分组成：蜂鸣器电路和单片机电路。下面开始进行系统的详细设计。

## 10.2 设计思路分析

蜂鸣器发声控制的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、I2C 控制器、定时器等多种外设，方便进行系统设计。

按照图 10-1 所确定的系统原理框图，需选择合适的蜂鸣器。为实现对蜂鸣器的控制，首先需了解蜂鸣器发声的原理。

### 10.2.1 蜂鸣器发声原理

**1. 电磁式蜂鸣器**

电磁无源蜂鸣器的工作原理是：交流信号通过绕在支架上的线包在支架的芯柱上产生一个交变的磁通，交变的磁通和磁环恒定磁通进行叠加，使膜片以给定的交流信号频率振动并配合共振腔发声。

电磁式有源蜂鸣器由振荡器、电磁线圈、磁铁、振动膜片及外壳等组成。接通电源后，振荡器产生的音频信号电流通过电磁线圈，使电磁线圈产生磁场。振动膜片在电磁线圈和磁铁的相互作用下，周期性地振动发声。

**2. 压电式蜂鸣器**

压电式无源蜂鸣器是以压电陶瓷为主要元件的。压电陶瓷是一类有将压力与电流相互转换能力的特殊陶瓷。这种能力源于其特殊的晶体结构。当压电陶瓷在一定方向上受到一个压力使其晶体结构发生形变时，它就会在内部产生一个电流，并且电流的变化与压力的变化密切相关，反之亦然。所以利用这一特性，在压电陶瓷上通过一定频率的电流，就会引起压电陶瓷微小的形变，这一形变带动空气发生振动，如果频率适当，就可以被人耳所听见，也就是产生了蜂鸣声。

压电式有源蜂鸣器主要由多谐振荡器、压电蜂鸣片、阻抗匹配器及共鸣箱、外壳等组成。有的压电式蜂鸣器外壳上还装有发光二极管。多谐振荡器由晶体管或集成电路构成。当接通电源后（1.5~15V 直流工作电压），多谐振荡器起振，输出一定频率的音频信号，阻抗匹配器推动压电蜂鸣片发声。压电蜂鸣片由锆钛酸铅或铌镁酸铅压电陶瓷材料制成。在陶瓷片的两面镀上银电极，经极化和老化处理后，再与黄铜片或不锈钢片粘在一起。

### 10.2.2 蜂鸣器发声控制

蜂鸣器的重要参数是正常发声频率，比如规格为 4kHz 的蜂鸣器，则是指该蜂鸣器发

声的频率是 4kHz。对发声的控制主要体现在 3 方面：

- 改变蜂鸣器控制端信号保持时间的长短，则可改变音长；
- 改变无源蜂鸣器控制端信号的频率，则可改变音调；
- 蜂鸣器控制端信号的有无保持可以形成蜂鸣器响声次数、音长、间隔的复杂控制。

## 10.3 硬件电路设计

通过图 10-1 可以看出，系统主要由电源电路、BDM 电路、蜂鸣器电路和单片机电路组成，下面重点介绍蜂鸣器电路和单片机电路。

### 10.3.1 蜂鸣器电路

蜂鸣器电路如图 10-2 所示，MC9S08JM60 的 PTA0 端口的控制信号经过三极管放大后驱动蜂鸣器发声。

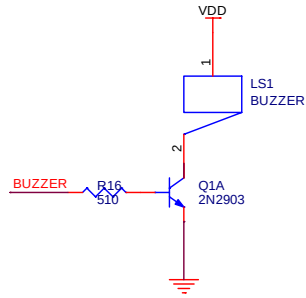



图 10-2 蜂鸣器电路图

### 10.3.2 单片机电路

本系统中，单片机通过内部的定时器产生蜂鸣器发声控制的时序，通过单片机的 I/O 口来控制蜂鸣器。如图 10-3 所示为具体的单片机电路图。

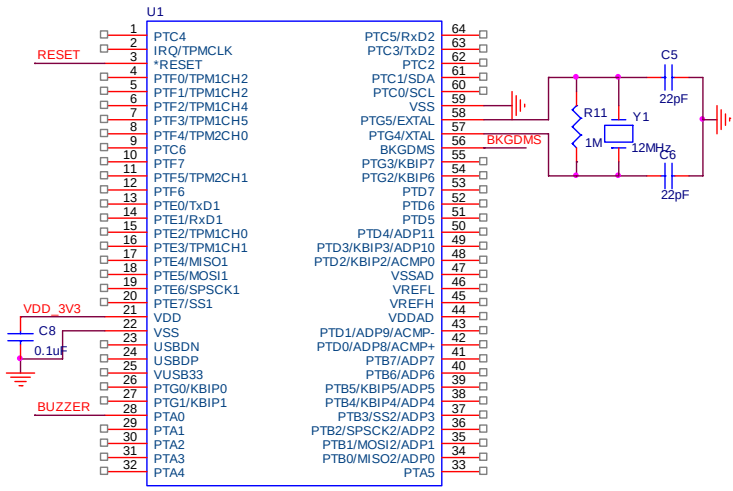



图 10-3 单片机电路图

## 10.4 软件设计与代码

### 10.4.1 软件设计流程

定时器控制蜂鸣器发声实例的系统软件比较简单。系统外设初始化完成后，开始变量

初始化，最后进入循环，控制蜂鸣器发声。系统软件的程序流程图如图 10-4 所示。

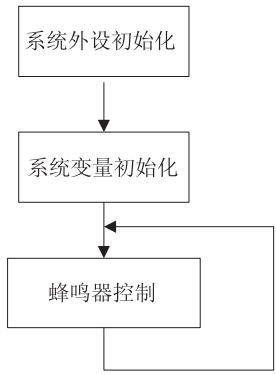



图 10-4　定时器控制蜂鸣器发声实例软件流程图

## 10.4.2　外设初始化程序

外设初始化程序主要包括端口初始化、定时器初始化等。下面主要介绍定时器初始化的具体程序。

本系统中，定时器使用的是内部定时器 TPM1。通过定时器初始化程序设置定时器工作的方式、定时中断的周期等，具体程序如下。

```
void timer_125us_Init (void)
{
  /* 定时器 1 状态和控制寄存器设置 */
  setReg8(TPM1SC, 0x00);
  /* 定时器 1 通道 0 状态和控制寄存器设置*/
  setReg8(TPM1C0SC, 0x50);
  /* 定时器 1 预置计数器寄存器和通道 0 计数器值设置 */
  timer_125us_SetCV(0x176FU);
  /* 定时器 1 计数寄存器清零*/
  setReg8(TPM1CNTH, 0x00);
  /* 定时器 1 状态和控制寄存器设置 */
  setReg8(TPM1SC, 0x08);
}
```

## 10.4.3　定时中断服务程序

定时器 TPM1 的定时中断周期为 125us，定时中断服务程序修改蜂鸣音长或间隔计数器，从而实现发声控制。下面为具体的定时中断服务程序。

```
void timer_125us_OnInterrupt(void)
{
  interrupt_cnt++;

  //每计 200 次中断，蜂鸣音长或间隔计数减 1
  if(interrupt_cnt == 200)
  {
    interrupt_cnt = 0;

    if(beep_delay !=0)
    {
      beep_delay--;
    }
  }

  if(time_f1.bit2 == 1)
   {
     PTAD ^=0x20;
  }
  else
  {
    PTAD = 0;
  }
}
```

### 10.4.4 蜂鸣器控制子程序

蜂鸣器控制子程序通过查询相关变量的值来控制单片机 PTA0 管脚的输出状态，从而驱动蜂鸣器发声与否。下面给出蜂鸣器控制子程序代码。

```
void Beep_Ctrl(unsigned char n)
{
   if(beep_delay == 0)
   {
     if(time_f1.bit2 == 1)
     {
        time_f1.bit2 = 0;
        PTAD = 0;
        beep_delay = beep_table[2 * (n-beep_cnt) + 1];
     }
     else
     {
         if(beep_cnt != 0)
         {
```

```
                time_f1.bit2 = 1;
                beep_cnt--;
                beep_delay = beep_table[2 * (n-beep_cnt)];
            }
        }
    }
}
```

### 10.4.5 主程序

系统经过一系列初始化后，即进入主程序工作。主程序调用蜂鸣器控制子程序，从而实现蜂鸣器的发声控制。下面给出实例的主程序代码。

```
void main(void)
{
    unsigned char h,l;
    PE_low_level_init();

    beep_delay = beep_table[0];
    beep_cnt = 0;
    n = beep_cnt;
    time_f1.bit2 = 1;

    for(;;)
    {
      Beep_Ctrl(n);
    }
}
```

## 10.5 实例总结

本章通过利用单片机内部定时器，实现了对蜂鸣器的发声控制。对蜂鸣器发声的控制有多种方法，本实例给出的方法通过对蜂鸣器控制信号的有无保持形成对蜂鸣器响声次数、音长、间隔的复杂控制。读者学完后，还可以在此基础上实现更复杂的设计，如实现音乐的节拍控制等。

# 第 11 章

# 定时器模块：电子钟系统设计实例

随着现代社会工作和生活节奏的加快，越来越需要人们对时间的准确把握。时间对人们来说总是那么宝贵，随时准确地知道时间并利用时间，是我们生活和工作中必不可少的。电子钟利用电子技术将时钟电子化、数字化，拥有时钟精确、体积小、界面友好、可扩展性能强等特点，是传统钟表的有力替代品。本章将介绍一个基于 MC9S08JM60 单片机实现的电子钟系统。首先介绍该系统的硬件设计，然后再介绍其软件设计。

## 11.1 实例内容说明

基于单片机控制的电子钟系统，根据基准信号产生的途径，可以分为两种方法：一是利用单片机中的定时器作为实时时钟基准；二是利用专用实时时钟日历芯片产生基准时钟信号。利用定时器实现的电子钟系统具有电路简单、成本低等特点，因此本章采用单片机内部的定时器作为实时时钟基准。基于单片机的电子钟系统主要功能是时间的显示，应能准确显示年、月、日、时、分、秒。同时还需有时间调整功能，电子钟可以根据当前时间进行时间调整。设置完成之后，电子钟在设定值基础上进行准确的计时和显示。电子钟还具有定时闹钟功能。可以通过键盘设定定时时间，这样当电子钟运行到设定的定时时间时，电子钟的蜂鸣器会发出语音提示，提示时间为一分钟。

电子钟系统的原理框图如图 11-1 所示。系统主要由按键电路、蜂鸣器电路、LCD 显示电路和 MC9S08JM60 构成。按键电路负责处理时间的设置和闹钟时间的设置，蜂鸣器电路主要用于闹钟功能，LCD 将当前时间显示出来。

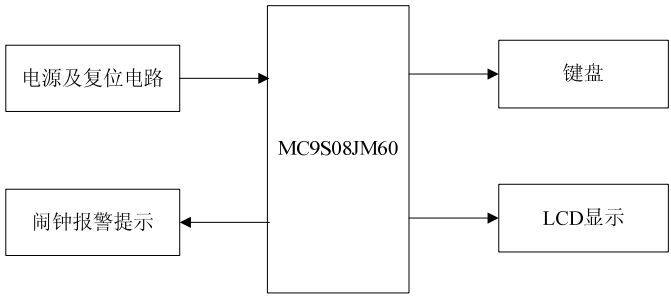



图 11-1　系统原理框图

## 11.2　设计思路分析

电子钟系统的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、I2C 控制器、定时器等多种外设，方便进行系统设计。

按照图 11-1 所确定的系统原理框图，选择合适的功能部件，以构成完整的系统控制电路。

**1. 键盘**

键盘是一组按键的集合，它是最常用的单片机输入设备。键盘可以分为两类：独立键盘和矩阵式键盘。

独立式键盘是最简单的键盘电路，每个键独立地接入一根数据线。平时所有的数据输入线都被连接成高电平，当任何一个键按下时，与之相连的数据输入线将被拉成低电平。要判断是否有键按下，只要用位处理指令即可。这种键盘的优点是结构简单，使用方便，但随着键数的增多占用的 I/O 口线也会增加。

矩阵式键盘，即通常所讲的行列式键盘，由行线和列线组成，按键位于行、列的交叉点上，行、列分别连接到按键开关的两端，列线通过上拉电阻接到高电平。无按键动作时，列线处于高电平状态；有按键按下时，交点的行线和列线接通，列线电平状态将由与此列线相连的行线电平决定。行线电平如果低，则列线电平为低；行线电平如果高，则列线电平也为高。矩阵式键盘节省了很多的 I/O 口，适用于按键数量比较多的场合。

在本系统中，由于时间设置的按键较多，除数字键 0～9 外，还有功能按键，为节省单片机 I/O 口线资源，我们选用 4×4 矩阵式键盘，占用了单片机 8 个 I/O 口线。

**2. LCD**

在单片机应用系统中，通常要进行信息显示。液晶显示器（LCD）由于具有很多优点，日益成为单片机应用系统的首选显示器件。

根据显示方式和内容的不同，液晶显示模块可以分为笔段型液晶显示模块、点阵字符液晶显示模块和点阵图形液晶模块 3 种。

笔段型液晶模块是一种由段型液晶显示器件与专用的集成电路组装成一体的功能部件，只能显示数字和一些标识符号；点阵字符液晶模块是由点阵字符液晶显示器件和专用的行列驱动器、控制器及必要的连接件、结构件装配而成的，可以显示字母、数字和符号，但不能显示图形；点阵图形液晶模块的点阵像素连续排列，行和列在排布中均没有空隔，不仅可以显示字符，而且可以显示连续、完整的图形。

显然，点阵液晶图形模块是 3 种液晶显示模块中功能最全面也最为复杂的一种。相比较，点阵字符液晶模块功能较均衡，使用也广泛。在本章的系统设计中，选用的就是 LCD1602 点阵字符液晶显示模块。

## 11.3 硬件电路设计

通过图 11-1 可以看出，系统主要由电源电路、闹钟报警提示电路、LCD 显示电路、按键电路和单片机电路组成，在本系统中，要求的电源电压为 3.3V。由于 3.3V 电源电路在第 5 章已经作了介绍，因此本章不再介绍，读者可以参看第 5 章的电源电路图。下面分别介绍闹钟报警提示电路、按键电路、LCD 显示电路和单片机电路。

### 11.3.1 闹钟报警提示电路

闹钟报警提示电路采用 I/O 通过三极管驱动蜂鸣器实现，由 MC9S08JM60 的 PTC4 管脚驱动，如图 11-2 所示。

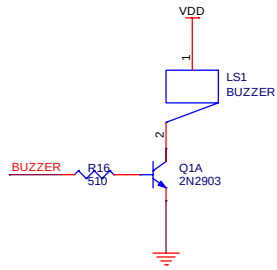



图 11-2 闹钟报警电路图

### 11.3.2 按键电路

电子钟需要通过键盘完成时间设置校准和闹钟设置。由于按键较多，这里采用了矩阵式键盘输入数据。如图 11-3 所示，MC9S08JM60 的 PTG0、PTG1、PTD2 和 PTD3 作为矩阵键盘的行控制线，配置为输出；PTD0、PTD1、PTD2 和 PTD3 作为矩阵键盘的列线输入，配置为输入，内部上拉使能，这样在没有按键按下时列线为高电平输入，当有按键按下时，对应的 I/O 口被拉低，这样检测 I/O 口电平状态即可判断按键按下与否。

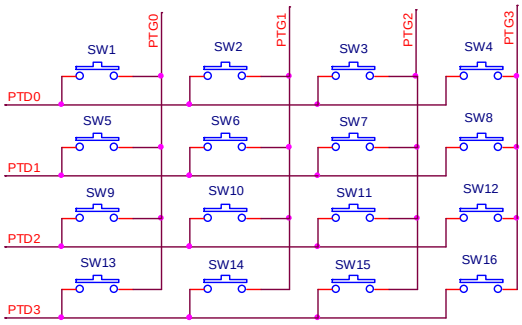



图 11-3 按键电路图

### 11.3.3 LCD 显示电路

LCD 电路用于电子钟的时间显示的实现，由于 MC9S08JM60 本身不带 LCD 控制器，因此我们选用内置 LCD 驱动器的 LCD 模块，如图 11-4 所示为 LCD 显示电路图。

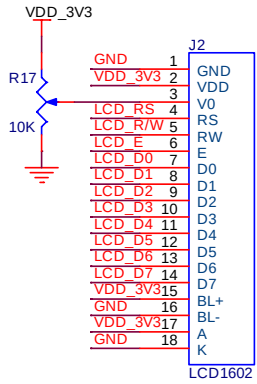



图 11-4 LCD 电路图

由图 11-4 可以看出，A、K 引脚用于背光，本例中为简化处理，直接接电源和地。数据线 D0～D7 和单片机的 PTE 口相连。RS 为寄存器选择，高电平时选择数据寄存器，低电平时选择指令寄存器。RW 为读写信号线，高电平时进行读操作，低电平时进行写操作，当 RS 和 RW 共同为低电平时可以写入指令或者显示地址，当 RS 为低电平且 RW 为高电平时可以读忙信号，当 RS 为高电平且 RW 为低电平时可以写入数据。E 端为使能端，当 E 端由高电平跳变成低电平时，液晶模块执行命令。控制线 RS、R/W 和使能线分别与单片机 PTC0～PTC2 引脚相连。

### 11.3.4 单片机电路

在电子钟系统中，单片机通过内部的定时器产生时钟基准，LCD 显示电路将时钟数据显示出来，由于 MC9S08JM60 本身不带 LCD 控制器，因此选用内置 LCD 控制器的 LCD 模块，通过单片机的 I/O 口来控制 LCD 模块。按键电路完成电子钟时间设置和闹钟设置。如图 11-5 所示为具体的单片机电路图。

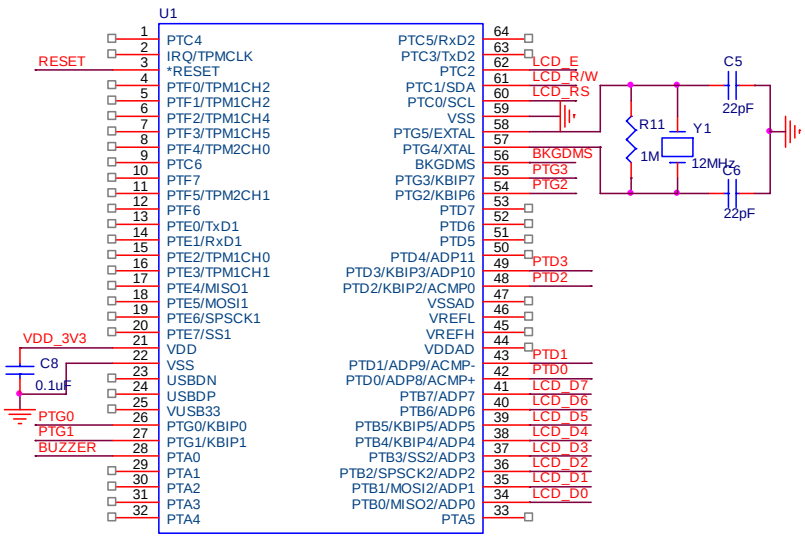



图 11-5 单片机电路图

如图 11-5 所示，MC9S08JM60 的 PTB 口连 LCD 的数据线，PTC0～PTC2 连 LCD 的控制线。PTD0～PTD3 连矩阵键盘的行线，PTG0～PTG3 连矩阵键盘的列线。

# 11.4 软件设计与代码

## 11.4.1 软件设计流程

电子钟系统的软件采用事件驱动的方式来编程，在主循环中不停地查询标志位，一旦某个标志位置为 1，则执行相应的子程序。电子钟系统软件流程图如图 11-6 所示。

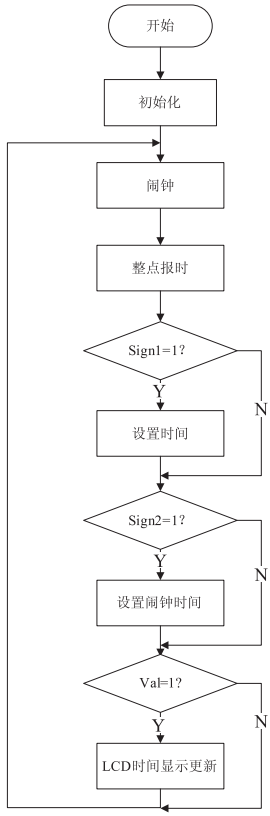



图 11-6 电子钟系统软件流程图

整个系统软件主要包括初始化程序、按键扫描程序、LCD 显示程序和主程序 4 个部分，下面介绍具体的程序。

## 11.4.2 初始化程序

初始化程序主要包括端口初始化、定时器初始化、键盘初始化和 LCD 初始化 4 个部分。主要设置相应的 I/O 管脚控制寄存器，以实现管脚的输入、输出和内部上拉等功能，键盘、LCD 和蜂鸣器等所用管脚的初始化程序也包括在内，以及定时器初始化等。具体程序如下。

```
void PE_low_level_init(void)
{
  /*I/O 端口初始化*/
  clrReg8Bits(PTAD, 0x01);
  clrReg8Bits(PTAPE, 0x01);
  setReg8Bits(PTADD, 0x01);
  clrReg8Bits(PTCD, 0x07);
  clrReg8Bits(PTCPE, 0x07);
  setReg8Bits(PTCDD, 0x07);
  setReg8(PTBD, 0x00);
  setReg8(PTBPE, 0x00);
  setReg8(PTBDD, 0x00);
  clrReg8Bits(PTDD, 0x0F);
  clrReg8Bits(PTDPE, 0x0F);
  setReg8Bits(PTDDD, 0x0F);
  setReg8Bits(PTGPE, 0x0F);
  clrReg8Bits(PTGDD, 0x0F);
  clrReg8Bits(PTASE, 0x3F);
  setReg8(PTBSE, 0x00);
  clrReg8Bits(PTCSE, 0x7F);
  setReg8(PTDSE, 0x00);
  setReg8(PTESE, 0x00);
  setReg8(PTFSE, 0x00);
  clrReg8Bits(PTGSE, 0x3F);
  setReg8(PTADS, 0x3F);
  setReg8(PTBDS, 0xFF);
  setReg8(PTCDS, 0x7F);
  setReg8(PTDDS, 0xFF);
  setReg8(PTEDS, 0xFF);
  setReg8(PTFDS, 0xFF);
  setReg8(PTGDS, 0x3F);
  Shadow_PTB = (byte)0x00;
  KEY_Init();                          //键盘中断初始化
  TIMER_1S_Init();                     //定时器初始化
  __EI();                             /* 开总中断*/
}
```

### 11.4.3 按键扫描程序

在矩阵式键盘扫描中，必须将行、列线信号配合起来作适当的处理，才能确定闭合键所在的位置。下面为具体的按键扫描程序。

```
Uchar KB_Scan(void)
 {
```

```
    Uchar line,i,tmp;
    line=0b11111110;              //使第一根行线为 0(低电平)
    for(i=1;i<=4;i++)             //将扫描 4 根行线
     {
      tmp=KEY_GetVal();            //取列线输入值
      tmp <<=4;
      tmp |= KEY_LINE_GetVal();  //取行线值
      tmp |= 0b00001111;           //行线全为 1
      KEY_LINE_PutVal(tmp&line);
       asm("NOP");                  //等待，消除按键抖动
       asm("NOP");
       tmp=KEY_GetVal();
       tmp&=0b11110000;           //看第 n 行是否有键按下
       if(tmp!=0xf0)              //本行有键按下
        {
          tmp=KEY_GetVal();
           break;
        }
        else                      //本行无键按下,扫描下一行
        {
           line=(line<<1)|0x01;
        }
    }
     if(i == 5)
     {
      tmp=0xff;
     }
     return(tmp);
}
```

### 11.4.4 LCD 显示程序

本系统中，选用内置 HD44780 驱动器的 1602 LCD 模块。HD44780 控制器的数据手册可去网上下载。LCD 在读写操作前，需要通过对 LCD 控制器写入一系列的命令来完成 LCD 控制器的初始化。下面给出 LCD 初始化的具体程序。

```
void  Lcd_Init(void)
{
   LCD_DATA_PutVal(0x30);
   LCD_CONTROL_ClrBit(0) ; //RS=0
   LCD_CONTROL_ClrBit(1) ; //R/W=0

   Toggle();
   WaitNms(5);
```

```
    LCD_DATA_PutVal(0x30);
    LCD_CONTROL_ClrBit(0) ; //RS=0
    LCD_CONTROL_ClrBit(1) ; //R/W=0
    Toggle();
    Wait1ms();

    LCD_DATA_PutVal(0x30);
    LCD_CONTROL_ClrBit(0) ; //RS=0
    LCD_CONTROL_ClrBit(1) ; //R/W=0
    Toggle();
    WaitNms(2);

    Lcd_Write_Command(0x38);//设置工作方式——8 位总线 字符行数两行 5*7 点阵
    Lcd_Write_Command(0x08);//设置显示状态——关显示
    Lcd_Write_Command(0x01);//清屏
    WaitNms(20);
    Lcd_Write_Command(0x06);//设置输入方式——画面不动光标右移
    Lcd_Write_Command(0x0F);//设置显示状态——画面显示光标、闪烁显示
}
```

对 LCD1602 的读写操作及屏幕和光标的操作都是通过对 HD44780 内部寄存器编程来实现的。下面给出 LCD 显示的子程序。

```
void Lcd_Show(Uchar str[])
{
    Uchar i;

    Lcd_Init();                 //Lcd 初始化
     Lcd_Write_Command(0x00);  //显示第 1 行
     for (i=0;i<10;i++)        //逐个显示
     {
         Lcd_Write_Data(str[i]);
     }

     Lcd_Write_Command(0xc0);   //显示第 2 行
    for (i=10;i<18;i++)         //逐个显示
    {
       Lcd_Write_Data(str[i]);
    }
}
```

### 11.4.5 主程序

系统经过一系列初始化后，即进入主程序工作。主程序调用按键扫描程序、LCD 显示程序等来完成电子钟的各种功能，主程序中标志位的改变在按键中断服务程序和定时器中断服务程序中。下面给出电子钟的主程序。

```
void main(void)
{
    Uchar h,l;;
    PE_low_level_init();

     val=0;
     sign1=0;
     sign2=0;

     Init_Time();                                    //初始化时间
     Time_Conver();
     Lcd_Show((Uchar *)time1);
     /*开总中断*/
     __EI();

    for(;;)
    {
      h=0;
     /*闹钟功能*/
      for(l=0;l<7;l++)
      {
         if(time[l]!=time2[l]) break;
         h++;
         if(h==7)
         {
          /*闹钟时间到，蜂鸣器响*/
           Alarm_Time(30);
         }
       }
       /*整点报时*/
      if((time[6]==0)&&(time[5]==0)&&(time[4]==0))
      {
        Buzzer_PutVal(1);
        Delay();
        Buzzer_PutVal(0);
        Delay();
      }
      if(sign1==1)
```

```
    {
      /*屏蔽键盘中断*/
       clrReg8Bits(KBIPE, 0xC3);
      /*设置时间*/
      Set_Time();
      sign1=0;
      /*打开键盘中断*/
      setReg8Bits(KBIPE, 0xC3);
    }
    if(sign2==1)
    {
      clrReg8Bits(KBIPE, 0xC3);
      sign2=0;
      /*设置闹钟时间*/
      Set_Alarm_Clock();
      setReg8Bits(KBIPE, 0xC3);
    }

    /*时间溢出中断发生 */
    if (val==1)
    {
        Time_Conver();
        Lcd_Show((unsigned char *)time1);
        val=0;
    }
  }
```

## 11.5 实例总结

本章通过利用单片机内部的定时器，辅之以矩阵键盘、LCD 显示和蜂鸣器等外围电路，设计出了简单、运行可靠的电子钟系统。LCD 模块选用最具代表性的内置 HD44780 驱动器的 1602，便于读者在其他系统设计中参考和移植。单片机采用的是 MC9S08JM60，它支持键盘中断，接口丰富，简化了程序的设计。本例设计方法与第 8 章的实时时钟设计有异曲同工之妙，读者可以用心体会。

# 第 12 章

# 存储器模块：FLASH 编程实现

在单片机的某些应用中，需要存放某些参数信息，一般都是通过外接 FLASH 来实现的，由于 MC9S08JM60 单片机有比较大的片内 FLASH 资源，因此可以利用单片机片内的 FLASH 存放参数信息。本章将介绍 MC9S08JM60 单片机的片内 FLASH 编程的实现过程。

## 12.1 实例内容说明

通过对 MC9S08JM60 单片机片内的 FLASH 编程的实现，就可以利用单片机片内的 FLASH 存放参数信息，从而省去外扩 EEPROM 或 FLASH 等存储器。如图 12-1 所示为系统的原理框图。

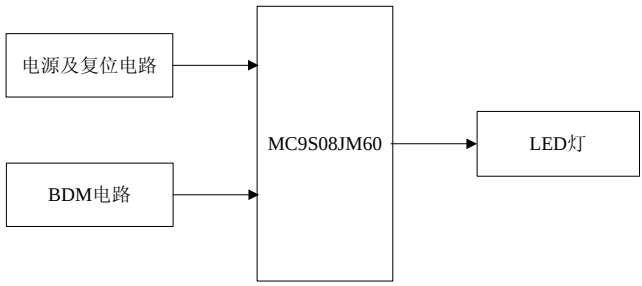



图 12-1 系统原理框图

由图 12-1 可以看出，系统的硬件电路比较简单，只是简单实现了 FLASH 操作功能，根据 LED 灯状态的变化提示操作成功与否。下面对具体的设计思路进行介绍。

# 12.2 设计思路分析

要实现 FLASH 的编程操作功能，首先需了解 MC9S08JM60 单片机片内 FLASH 的相关操作寄存器和 FLASH 编程的步骤。下面先介绍 FLASH 操作的相关寄存器。

## 12.2.1 寄存器介绍

实现 FLASH 操作的寄存器主要有：FCDIV、FOPT、NVOPT、FCNFG、FPROT、NVPROT、FSTAT 和 FCMD。下面对主要寄存器具体进行介绍。

**1. FLASH时钟分频寄存器FCDIV**

该寄存器的第 7 位是一个只读位，6：0 位可以在任何时候读取但只能写入一次。在开始任何擦除或编程操作之前，写入该寄存器将非易失性内存系统的时钟频率设置在可接受的限度内。

FCDIV 寄存器的各控制位如下所示。

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | DIVLD | PRDIV8 | DIV5 | DIV4 | DIV3 | DIV2 | DIV1 | DIV0 |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

DIVLD 为预分频器加载状态标识位。

1=FCDIV 自复位后已写入，允许 FLASH 和 EEPROM 擦除和编程操作。

0=FCDIV 自复位后没有写入，禁止 FLASH 和 EEPROM 擦除和编程操作。

PRDIV8 为 FLASH 时钟 8 预分频位。

1=FLASH 时钟输入为总线时钟的 8 分频。

0=FLASH 时钟为总线时钟。

DIV[5:0]为 FLASH 时钟的分频因子位。最后的内部 FLASH 时钟的频率必须在 200kHz 到 150kHz 的范围内，这样才能使 FLASH 和 EEPROM 正常运行。

FLASH 的时钟由 PRDIV8 位和 DIV[5：0]位共同确定。计算公式为：

如果 PRDIV8=0，则 FLASH 时钟 fFCLK=总线时钟/（DIV[5:0]+1）

如果 PRDIV8=1，则 FLASH 时钟 fFCLK=总线时钟/（8×（DIV[5:0]+1））

**2. FLASH选项寄存器FOPT**

系统上电复位过程中，在非易失位置中 NVOPT 的内容从 FLASH 复制到 FOPT。第 2~5 位不使用并总是读出 0。在任何时间这个寄存器都可以被读出，但写入没有意义或影响。要改变这个寄存器的值，照常擦除和重新编程 FLASH 存储器中 NVOPT 位置，然后发起一个新的 MCU 复位。

FOPT 寄存器的各控制位如下所示。

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | KEYEN | FNORED | 0 | 0 | 0 | 0 | SEC01 | SEC00 |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

KEYEN 为后门密钥机制允许位。

1=如果用户固件写入的 8 位值匹配非易失的后门密钥，在下一次 MCU 复位前，安全保护暂时失效。

0=禁止后门密钥访问。

FNORED 为矢量变向允许位。

1=允许向量重映射

0=禁止向量重映射

SEC0[1:0]为保密状态码位。

该两位决定 MCU 的保密状态，如表 12-1 所示。当 MCU 处于保密状态，RAM 和 FLASH 存储器中的内容不能通过从任何非保密源的包括背景调试接口的指令来访问。

表 12-1　SEC0[1:0] 保密状态码位描述

| SEC01:SEC00 | 描　　述 |
|---|---|
| 0:0 | 安全 |
| 0:1 | 安全 |
| 1:0 | 不安全 |
| 1:1 | 安全 |

**3. FLASH配置寄存器FCNFG**

在任何时候位 7:5 可以被读或写；位 4:0 总是读出 0 但不能写入。FCNFG 寄存器的各控制位如下所示。

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | 0 | 0 | KEYACC | 0 | 0 | 0 | 0 | 0 |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

KEYACC 为写访问密钥允许位：该位使能写后门比较钥匙。

1=写 NVBACKKEY（$FFB0～$FFB7）被解释为写比较钥匙。

0=写地址 0XFFB0～0XFFB7 被解释为启动 FLASH 编程或擦除操作。

**4. FLASH保护寄存器FPROT**

在复位过程中，非易失位置 NVPROT 的内容从 FLASH 复制到 FPROT。在任何时间

这个寄存器可被读。

FPROT 寄存器的各控制位如下所示。

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | FPS7 | FPS6 | FPS5 | FPS4 | FPS3 | FPS2 | FPS1 | FPDIS |
| | 写 | | | | | | | | |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FPS[7:1]为 FLASH 保护选择位。当 FPDIS=0，这个 7 位的域确定了在 FLASH 结束的高地址的非保护 FLASH 位置的结束地址。

FPDIS 为 FLASH 保护允许位。

1=没有 FLASH 块被保护。

0=FPS[7:1]指定 FLASH 块保护的区域（不被允许的编程和擦除）。

**5. FLASH状态寄存器FSTAT**

位 3、1、0 总是读出 0，写入操作没有意义或影响。其余 5 位是状态标志位，可以在任何时间被读出。

FSTAT 寄存器的各控制位如下所示。

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | FCBEF | FCCF | FPVIOL | FACCERR | 0 | FBLANK | 0 | 0 |
| | 写 | | | | | | | | |
| 复位 | | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

FCBEF 为 FLASH 命令缓冲器空标志位。该位用来开始命令。当执行突发编程时，这表明了命令缓存为空以致后面的命令能被执行。

1=一个新的突发编程命令可以被写入到编程缓存。

0=命令缓冲器满（没准备好另外的命令）。

FCCF 为 FLASHA 命令完成标志位。当命令缓存为空且没有正在执行的命令时，该标志位被自动置 1。启动一个新命令时，FCCF 标志自动清零。写 FCCF 没有意义和影响。

1=所有命令已完成。

0=命令正在操作中。

FPVIOL 为保护冲突标志位。当 FCBEF 为登记一个试图擦除或者编程一个受保护块位置的命令而清零时，该位自动置 1。该位通过对其写 1 被清除。

1=试图擦除或编程一个受保护区域。

0=未发现保护冲突。

FACCERR 为访问错误标志位。当没有遵守正确的命令顺序，如果在 FCDIV 寄存器初始化之前试图进行一个编程或者擦除操作，或者命令正在进行时 MCU 进入停止状态，该位自动置 1。

1=产生一个访问错误。

0=没有访问错误。

FBLANK 为 FLASH 空校验标志位：当全部 FLASH 被确认为擦除，在一个空检查完成时，该位自动置 1。该位通过清除 FCBEF 写新的有效的命令来清除。

1=在一个空检查命令之后且 FCCF=1，FBLANK=1 表示 FLASH 已经完成擦除了。

0=在一个空检查命令之后且 FCCF=1，FBLANK=0 表示 FLASH 没有完成擦除。

**6. FLASH命令寄存器FCMD**

FCMD 寄存器的各控制位如下所示。

| 数据位 | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| 定义 | 读 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 写 | FCMD7 | FCMD6 | FCMD5 | FCMD4 | FCMD3 | FCMD2 | FCMD1 | FCMD0 |
| 复位 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FCMD[7:0]为 FLASH 命令位。

FLASH 命令和 FLASH 命令位的关系如表 12-2 所示。

表 12-2　FLASH 命令和 FLASH 命令位的关系

| 命令 | FCMD |
|---|---|
| 空检查 | 0x05 |
| 字节编程 | 0x20 |
| 字节编程—突发模式 | 0x25 |
| 页擦除（512 字节/页） | 0x40 |
| 整片擦除 | 0x41 |

## 12.2.2　FLASH 编程和擦除命令的执行顺序

为正确操作 FLASH 存储器，需了解 FLASH 操作命令的执行步骤。在执行命令之前任何错误标志都需要被清除。命令执行的步骤是：

①将一个数据值写入到 FLASH 阵列中的一个地址中。该地址和写入的数据信息被锁定到 FLASH 接口上。这一写入操作是任何命令序列中要求的第一步。对于擦除和空白检查命令，这些数据的值并不重要。对于页擦除命令，地址可以是将要擦除的 FLASH512 字节页面中的任何地址。对于整体擦除和空检查命令，地址可以是 FLASH 存储器中的任何地址。512 字节的整个页面是 FLASH 能被擦除的最小块。在 60KB 的版本中，有两个实例，用户可用的块的大小少于 512 字节： RAM 以下的第一页和在高页寄存器下的第一页。这些页分别被 RAM 和高页寄存器页重叠。

②将命令代码写入到 FCMD 中。 6 个有效的命令分别是空白检查（$05）、字节编程

（$20）、突发编程（$25）、分区擦除（$40）、整体擦除（$41）和分区擦除终止（$47）。命令代码被锁定到命令缓冲器中。

③将一个 1 写入到 FSTAT 中的 FCBEF 位上，以清除 FCBEF 并发起命令 （包括其地址和数据信息）。

在写内存阵列之后到写 1 用于清除 FCBEF 并发起完整命令之前的任何时候，可以通过向 FCBEF 中写入一个“0”，来手动终止部分命令顺序。以这种方式终止一个命令会设置 FACCERR 访问错误标记，而这个标记必须在开始一个新命令之前被清除。

整个过程必须遵守严格的监控流程，否则命令将不会被接受。命令完整标记（FCCF）用于指示一条命令是否完整。要启动命令，必须通过清除 FCBEF 来使命令序列完整。在使用任何 FLASH 命令之前，FCDIV 寄存器必须初始化。复位后，这只能执行一次。

## 12.3 硬件电路设计

本系统的硬件接口电路比较简单，主要是单片机电路的搭建。单片机的电路比较简单，只需要单片机的 PTB7 与 LED 等相连。另外还需要相应的振荡电路才能使单片机进行工作，如图 12-2 所示为具体的单片机电路图。

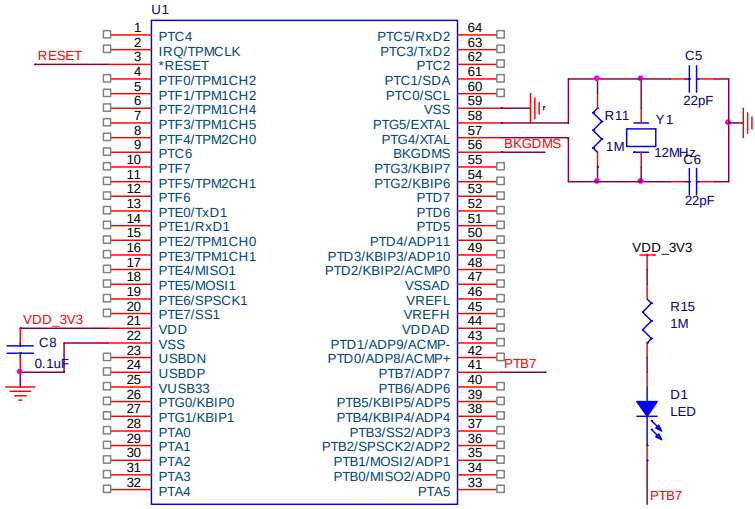



图 12-2 单片机电路图

## 12.4 软件设计与代码

### 12.4.1 软件设计流程

FLASH 实例的软件比较简单，系统经过初始化后，开始读写 FLASH 操作，操作出错则进行状态提示。系统的软件流程图如图 12-3 所示。

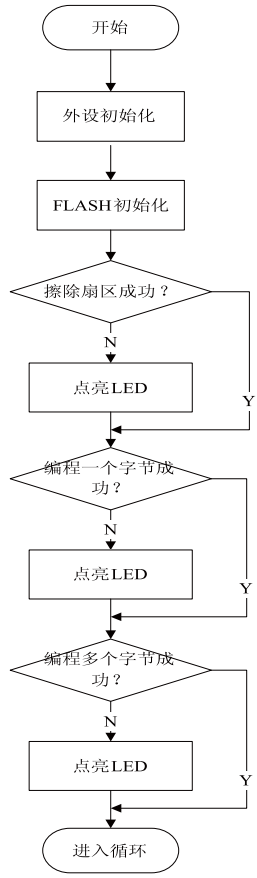



图 12-3　软件流程图

下面介绍 FLASH 操作的子程序。

## 12.4.2　FLASH 操作子程序

FLASH 的操作主要包括读、写和擦除操作。其读操作非常简单，只需利用 C 语言的指针就可以实现指定地址的读数据操作。下面给出 FLASH 写操作和擦除操作的具体程序。

```
unsigned char Flash_Cmd(unsigned int FlashAddress, unsigned int
                        FlashDataCounter, unsigned char* pFlashDataPtr,
                        unsigned char FlashCommand)
{
   /* 标志位 FACCERR 或 PVIOL 是否被置 1 */
    if (FSTAT&0x30)              //如果标志位被置 1，则清零
    {
        FSTAT = FSTAT | 0x30;
    }

    if (FlashDataCounter)
```

```
    {
      do
      {
          /* 等待最后一个突发命令完成*/
          while(!(FSTAT&0x80));

          /* 向 FLASH 中写数据*/
          (*((volatile unsigned char *)(FlashAddress++))) = *pFlashDataPtr;
          pFlashDataPtr++;
          /* 写命令 */
          FCMD = FlashCommand;
          /* Put FCBEF at 1 */
          FSTAT = 0x80;

          /* 延时 4 个周期*/
          _asm NOP;
          _asm NOP;
          _asm NOP;
          _asm NOP;

          /* 检查 FLASH 访问出错标志位或保护冲突标志位是否被置 1 */
          if (FSTAT&0x30)
          {
            /* 如果被置 1，则函数返回 1，表示错误 */
            return (1);
          }
      }while (--FlashDataCounter);
    }
    /* 等待最后一个命令完成 */
    while ((FSTAT&0x40)==0);
    /* 返回值为 0，表示函数执行正确 */
    return (0);
}

#define Flash_Erase(Address) \
        Flash_Cmd((unsigned int)Address, (unsigned int)1,
                  (unsigned char*) &FlashBuffer[0], FLASH_ERASE_CMD)

#define Flash_Program(Address, Data) \
        Flash_Cmd((unsigned int)Address, (unsigned int)1,
                  (unsigned char*)&Data, FLASH_PROGRAM_CMD)
```

### 12.4.3 主程序

实例经过外设初始化后，将 FLASH 中的程序搬到 RAM 中运行，然后 FLASH 初始化，接着擦除一个扇区、编程一个字节、编程多个字节，若操作 FLASH 过程中出现错误，则

点亮 LED 状态指示灯。下面给出实例的主程序。

```
void main(void)
{
  unsigned char Data = ' ';
  /*系统初始化*/
  PE_low_level_init();
   /*将 FLASH 中的程序复制到 RAM 中 */
  CopyInRAM();

  /* 当发生 FLASH 操作错误时，点亮 LED 灯 */
  PTBDD |= 0x80;
  PTBD |= 0x80;
  /*FLASH 模块初始化*/
  FlashInit();
  /*擦除一个扇区 */
  if (Flash_Erase(0xFA00))
  {
    PTBD &= ~0x80;
  }

  /*编程一个字节*/
  if (Flash_Program(0xFA00, Data))
  {
    PTBD &= ~0x80;
  }

  /*编程一个字符串 */
  if (Flash_Burst(0xFA01, sizeof(FlashBuffer), &FlashBuffer[0]))
  {
    PTBD &= ~0x80;
  }

  for(; ;) {
    WDog1_Clear();
  }
}
```

## 12.5 实例总结

本章介绍了 MC9S08JM60 片内 FLASH 模块的操作，给出了 FLASH 操作的具体程序。通过对 FLASH 的操作，用户可以使用片内 FLASH 来存放参数信息。从本章的设计中读者可以体会到 FLASH 模块编程的一般原理。

# 第 13 章

# AD 模块：加速度传感器系统设计实例

在汽车仪表应用领域和振动检测领域，加速度传感器是一个重要的部件。利用加速度传感器可以很方便地将加速度这一物理信号转变成便于测量的电信号。本章将介绍基于 MC9S08JM60 单片机的加速度传感器系统的实现。

## 13.1 实例内容说明

一般加速度传感器是利用了其内部的由于加速度造成的晶体变形这个特性。由于这个变形会产生电压，只要计算出产生电压和所施加的加速度之间的关系，就可以将加速度转化成电压输出。当然，还有很多其他方法来制作加速度传感器，例如压阻技术、电容效应、热气泡效应、光效应，但是其最基本的原理都是由于加速度使某个介质产生了变形，通过测量其变形量并用相关电路转化成电压而输出。如图 13-1 所示为系统的原理框图。

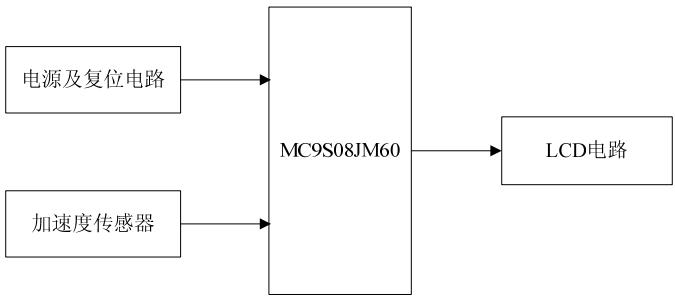



图 13-1　系统原理框图

由图 13-1 可以看出，系统的硬件电路比较简单，由加速度传感器将加速度信号转换成电压信号，送入 MC9S08JM60 单片机的 AD 接口，经 AD 转换处理后，结果显示在 LCD 上。

## 13.2 设计思路分析

加速度传感器系统的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、I2C 控制器、定时器等多种外设，方便进行系统设计。

分析图 13-1 所确定的系统原理框图，加速度传感器是整个系统的关键，因此首先需选择合适的加速度传感器芯片。

加速度传感器是一种能够测量加速力的电子设备。加速力就是当物体在加速过程中作用在物体上的力，就好比地球引力，也就是重力。加速力可以是个常量，比如 g，也可以是变量。

加速度计有两种：一种是角加速度计，是由陀螺仪（角速度传感器）而改进的；另一种就是线加速度计。

在选择加速度传感器时，主要需考虑以下技术指标：

- 模拟输出 vs 数字输出：
  这个是最先需要考虑的。这个取决于你系统和加速度传感器之间的接口。一般模拟输出的电压和加速度是成比例的，比如 2.5V 对应 0g 的加速度，2.6V 对应于 0.5g 的加速度。数字输出一般使用脉宽调制（PWM）信号。
  如果你使用的微控制器只有数字输入，那你就只能选择数字输出的加速度传感器了；如果你使用的微控制器有模拟输入口，你可以非常简单地使用模拟接口的加速度传感器。
- 测量轴数量：
  对于多数项目来说，两轴的加速度传感器已经能满足多数应用了。对于某些特殊的应用，比如 UAV、ROV 控制，三轴的加速度传感器可能会更适合一点。
- 最大测量值：
  如果你只要测量机器人相对于地面的倾角，那一个±1.5g 加速度传感器就足够了。但是如果你需要测量机器人的动态性能，±2g 也应该足够了。要是你的机器人会有突然启动或者停止的情况出现，那你需要一个±5g 的传感器。
- 灵敏度：
  一般来说，越灵敏越好。越灵敏的传感器对一定范围内的加速度变化更敏感，输

出电压的变化也越大，这样就比较容易测量，从而获得更精确的测量值。

- 带宽：
  这里的带宽实际上指的是刷新率，也就是说传感器每秒钟会产生多少次读数。对于一般只要测量倾角的应用，50Hz 的带宽应该足够了，但是对于需要进行动态性能的测量，比如振动，你会需要一个具有上百赫兹带宽的传感器。
- 电阻/缓存机制：
  对于有些微控制器来说，要进行 A/D 转化，其连接的传感器阻值必须小于 10kΩ。例如加速度传感器的阻值为 32kΩ，在 PIC 和 AVR 控制板上无法正常工作，所以建议在购买传感器前，仔细阅读控制器手册，确保传感器能够正常工作。

经过综合权衡，我们在此选用 Freescale 公司的输入轴加速度传感器 MMA1220D 和 MMA1260D 作为本次设计的加速度传感器芯片。

## 13.3 硬件电路设计

加速度传感器系统设计实例主要由电源电路、LCD 显示电路、加速度传感器电路和单片机电路组成。在本系统中，要求的电源电压为 3.3V，由于 3.3V 电源电路在第 5 章已经作了介绍，因此本章不再介绍，读者可以参看第 5 章的电源电路图。下面分别介绍加速度传感器电路、LCD 显示电路和单片机电路。在介绍具体的电路之前，首先对 MMA1220D 和 MMA1260D 进行简单的介绍。

### 13.3.1 MMA1220D 和 MMA1260D 的介绍

MMA1220D 和 MMA1260D 都是 Freescale 公司生产的线性加速度传感器芯片，二者主要特性基本相同，但 MMA1220D 功能更强，测量范围更宽。MMA1220D 和 MMA1260D 的主要功能特性如下：

- 集成内部信号调理。
- 线性输出。
- MMA1220D 内置 4 阶巴塞尔滤波器；MMA1260D 内置 2 阶巴塞尔滤波器。
- 内部自校正功能。
- MMA1220D 内置低电压检测、时钟监视和 EPROM 奇偶检查状态；MMA1260D 只内置 EPROM 奇偶检查状态。
- 变送器密封设计确保高可靠性。
- 突发设计，具有高冲击生存能力。
- 宽测量范围：MMA1220D 测量范围为±8g；MMA1260D 测量范围为±1.5g.
- 供电电压为：0.3V~7.0V。

为便于进行硬件电路的设计，下面给出芯片 MMA1220D 和 MMA1260D 的管脚图，如图 13-2 和图 13-3 所示。

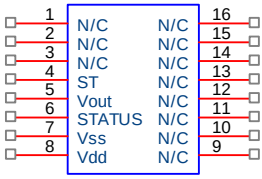



图 13-2　MMA1220D 管脚图

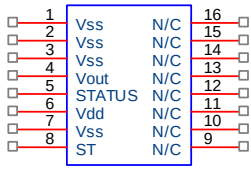



图 13-3　MMA1260D 管脚图

由图 13-2 和图 13-3 可知，MMA1220D 和 MMA1260D 都为 16 个管脚，使用起来比较简单，只需要简单的外围电路。下面对具体的管脚进行介绍：

- NC：空引脚，悬空不使用。
- VDD：可选电源管脚，电源电压范围为 0.3~7.0V。
- ST：逻辑输入管脚，用以启动芯片自测试。
- VOUT：加速度传感器的输出电压。
- STATUS：指示故障的逻辑输出管脚。
- VSS：接地管脚。

## 13.3.2　加速度传感器电路

加速度传感器的电路如图 13-4 所示。加速度传感器 MMA1220D 的输出电压接 MC9S08JM60 的 PTD4 口，故障逻辑输出管脚 STATUS 接 MC9S08JM60 的 PTD6 口；加速度传感器 MMA1260D 的输出电压接 MC9S08JM60 的 PTD3 口，故障逻辑输出管脚 STATUS 接 MC9S08JM60 的 PTD5 口。MMA1220D 和 MMA1260D 的逻辑输入管脚 ST 接 MC9S08JM60 的 PTD7 口。

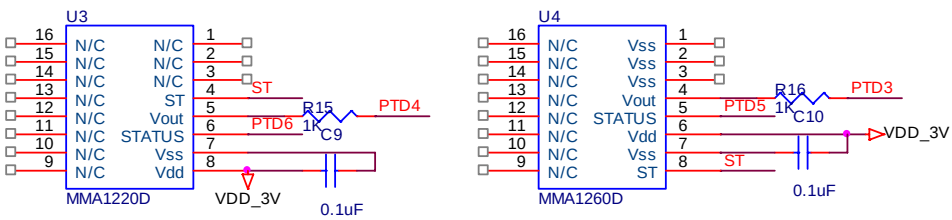



图 13-4　加速度传感器电路图

## 13.3.3　LCD 显示电路

LCD 电路用于显示测量的加速度，由于 MC9S08JM60 本身不带 LCD 控制器，因此我们选用内置 LCD 驱动器的 LCD 模块，如图 13-5 所示为 LCD 显示电路图。

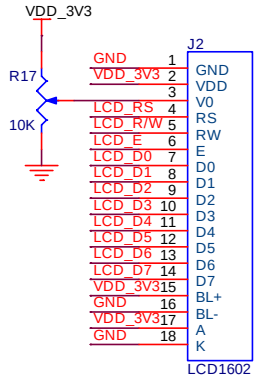



图 13-5　液晶显示电路图

如图 13-5 可以看出，A、K 引脚用于背光，本例中为简化处理，直接接电源和地。数据线 D0～D7 和单片机的 PTB 口相连。RS 为寄存器选择，高电平时选择数据寄存器，低电平时选择指令寄存器。RW 为读写信号线，高电平时进行读操作，低电平时进行写操作，当 RS 和 RW 共同为低电平时可以写入指令或者显示地址，当 RS 为低电平且 RW 为高电平时可以读忙信号，当 RS 为高电平且 RW 为低电平时可以写入数据。E 端为使能端，当 E 端由高电平跳变成低电平时，液晶模块执行命令。控制线 RS、R/W 和使能线分别与单片机 PTC0～PTC2 引脚相连。

## 13.3.4　单片机电路

单片机的电路比较简单，只需要单片机的 PTB0～PTB7、PTC0～PTC2、PTD3～PTD7 与 LCD 显示电路、加速度传感器电路接口。另外单片机还需相应的振荡电路才能进行工作，如图 13-6 所示为具体的单片机电路图。

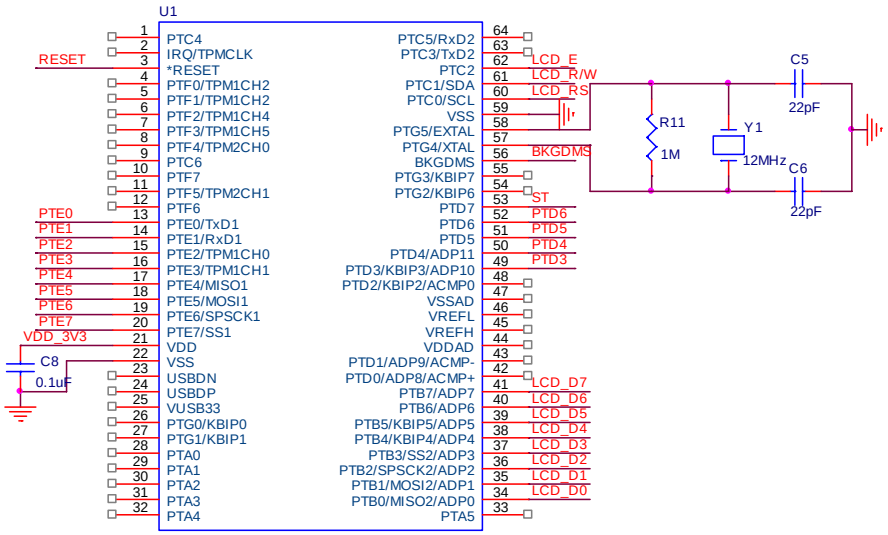



图 13-6　单片机电路图

# 13.4 软件设计与代码

## 13.4.1 软件设计流程

加速度传感器系统的软件程序比较简单，主要包括 AD 采样子程序、加速度传感器子程序、LCD 显示子程序和主程序 4 个部分。主程序是个典型的无限循环系统，LCD 不断刷新显示最新的加速度值。主程序流程如图 13-7 所示，上电复位后，系统初始化完成后，将测试加速度传感器 MMA1220D 和 MMA1260D，测试完成后，则进入主循环，启动 A/D 转换，将 A/D 转换的值处理后显示在 LCD 上。下面分别介绍具体的程序。

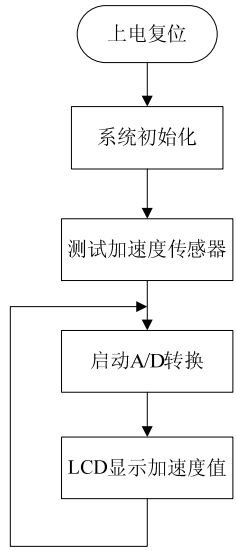



图 13-7 主程序流程图

## 13.4.2 A/D 采样子程序

A/D 采样子程序采用中断方式，转换的结果放入全局缓冲区 AD1_OutV[2]中。主程序通过调用 AD1_GetValue16（）函数获取 A/D 转换的结果。下面介绍具体的程序。

```
ISR(AD1_Interrupt)
{
  ((TWREG*)(&AD1_OutV[SumChan]))->b.high = ADCRH; /* 保存测量值 */
  ((TWREG*)(&AD1_OutV[SumChan]))->b.low = ADCRL;
  SumChan++;                          /* A/D 通道总数加 1*/
  if (SumChan == 2) {                  /* A/D 通道总数是否为 2？ */
    SumChan = 0;                      /*若是 2，则清零 A/D 通道总数 */
    OutFlg = TRUE;                    /* 测量结果可读标志置 1 */
    AD1_OnEnd();                       /* 调用用户事件*/
    ModeFlg = STOP;                   /* 设置设备为停止模式 */
    return;
  }
  ADCSC1 = Channels[SumChan];          /* 启动下一 A/D 通道转换 */
}
byte AD1_GetValue16(word *Values)
{
if (!OutFlg)                         /* 输出标志是否置位*/
{
    return ERR_NOTAVAIL;              /* 未置位，则返回错误状态*/
  }
  Values[0] = (word)((AD1_OutV[0]) << 4); /*将测量值放入输出缓冲区中*/
  Values[1] = (word)((AD1_OutV[1]) << 4);
  return ERR_OK;                      /* 返回正确状态*/
}
```

### 13.4.3 加速度传感器子程序

加速度传感器子程序主要包括 3 个子程序：AccTest（）、AccDataConvGs（）和 AccDataFormat（）。AccTest（）函数主要测试加速度传感器的状态，测试完成的状态显示在 LCD 上；AccDataConvGs（）函数将 A/D 的转换值转化为加速度的单位 g，以供 LCD 显示；AccDataFormat（）函数将 A/D 的转换值处理后送入 LCD 的消息缓冲区中。下面给出加速度传感器的子程序。

```
void AccTest(Uchar u8AccUnderTest)
{
    bool bTestStatus = FAIL;
    Uchar u8SelfTestCount = 0;
    Uchar u8Value1 = 0;
    Uchar u8Value2 = 0;

    Lcd_Write_Command(CLEAR_LCD);
    Lcd_Write_Command(CURSOR_HOME);

    Lcd_Disp_Str(0x00, 0x00, "Testing ");   //加速度计正在测试

    if(u8AccUnderTest == MMA1220)
    {
        Lcd_Disp_Str(0x00, 0x08, "MMA1220 ");
    }
    else if(u8AccUnderTest == MMA1260)
    {
      Lcd_Disp_Str(0x00, 0x08, "MMA1260 ");
    }

    do
    {
       ST_ClrVal();                              //自测试引脚拉低
       WaitNms(25);

         u8Value1 = AD1_GetValue16(u8AccUnderTest); //启动 A/D 转换
       ST_SetVal() ;                       //自测试引脚拉高
       WaitNms(25);

       u8Value2 = AD1_GetValue16(u8AccUnderTest);

       if(u8Value2 > u8Value1)    // 输出变化是否为正值?
       {
         u8Value2 -= u8Value1;
```

```
        /*检查变化范围是否处于加速度计指标范围内*/
        if((u8AccUnderTest == MMA1220) && (u8Value2 > MMA1220_MIN_DELTA)
 && (u8Value2 < MMA1220_MAX_DELTA) )
        {
                bTestStatus = OK;
                break;
        }
        if((u8AccUnderTest == MMA1260) && (u8Value2 > MMA1260_MIN_DELTA)
 && (u8Value2 < MMA1260_MAX_DELTA) )
        {
                bTestStatus = OK;
                break;
        }

      }

      u8SelfTestCount++;           //测试计数器加 1
   }while(u8SelfTestCount<5);

   ST_ClrVal();
   Lcd_Disp_Str(0x01, 0x00, "Self-Test");

   if (bTestStatus == OK)
   {
     Lcd_Disp_Str(0x01, 0x0B, "OK"); //自测试成功
   }
   else
   {
     Lcd_Disp_Str(0x01, 0x0B, "Failed"); // 自测试失败
   }
   WaitNms(250);                                    //延时 0.5ms
   WaitNms(250);
   return;
}

Uchar AccDataConvGs(Uchar u8RawValue)
{
    Uint u16FormattedValue;

    if(gu8AccSelect == MMA1220)
{
        /*将 MMA1220 的 A/D 转换值转化成单位 Gs*/
         u16FormattedValue = u8RawValue*100;
         u16FormattedValue /= MMA1220_CONST;
```

```
    }
    else if(gu8AccSelect == MMA1260)
{
     /*将 MMA1260 的 A/D 转换值转化成单位 Gs*/
         u16FormattedValue = u8RawValue*10;
         u16FormattedValue /= MMA1260_CONST;
}
    return((Uchar)u16FormattedValue);
}

bool AccDataFormat(Uchar u8AccData)
{
    Uchar u8BufferPosition;
    bool bNewTopFlag = FALSE;

if(u8AccData < 128)     //数据是负值
    {
       gau8LcdFirstLine[9] = '-';
       u8AccData = 128 - u8AccData;         // Remove offset
    }
    else                    //数据是正值
    {
       gau8LcdFirstLine[9] = ' ';
       u8AccData = u8AccData - 128;         // Remove offset
    }

    if((u8AccData > gu8TopAccValue) || (gbResetTop))
    {
       gu8TopAccValue = u8AccData;          // 更新最大值
       bNewTopFlag = TRUE;
       gbResetTop = FALSE;
       gbOveraccState = FALSE;
    }

    u8AccData = AccDataConvGs(u8AccData);   // 将数据转换成 Gs

    if(gu8AccSelect == MMA1220)            // 将变化标志设为 MMA1220
    {
       gau8LcdFirstLine[5] = '2';
       if(u8AccData>80) gbOveraccState = TRUE;
    }

    if(gu8AccSelect == MMA1260)         // 将变化标志设为 MMA1260
    {
```

```
        gau8LcdFirstLine[5] = '6';
        if(u8AccData>15) gbOveraccState = TRUE;
      }

      if(u8AccData<10)
      {
        gau8LcdFirstLine[10]='0';      // 如果 G<1，则加 0
      }

      u8BufferPosition = 12;                    //将 BCD 转换后送入信息缓冲区中
      do
      {
        gau8LcdFirstLine[u8BufferPosition--]= (u8AccData%10) + '0';
        u8BufferPosition--;
      }while((u8AccData /= 10) > 0);

      return(bNewTopFlag);                              // 返回最大值
  }
```

### 13.4.4 LCD 显示程序

LCD 显示程序将 A/D 转换处理后的加速度值显示在 LCD 上。具体的 LCD 显示子程序可参考第 12 章中的 LCD 显示子程序部分。

### 13.4.5 主程序

系统经过一系列初始化后，即进入主程序工作。主程序调用 A/D 采样子程序、加速度传感器子程序、LED 显示程序等来完成加速度传感器系统，下面给出主程序代码。

```
  void main(void)
  {
     Uchar au8Lcd2ndLine[]="MAX:   . G  --- ";
     Uchar u8NewAdcValue;
     bool bNewTopValue;

      /* 先读 MMA1220 转换值*/
      gu8AccSelect = MMA1220;

    /系统初始化*/
    PE_low_level_init();
    InitHardware();

     /* 在 LCD 上显示欢迎信息 */
      Lcd_Disp_Str(0x00, 0x05, "ACCELEROMETER");
```

```
    Lcd_Disp_Str(0x01, 0x05, "EVALUATION BOARD");

   /* 延时 0.75s */
    WaitNms(250);
    WaitNms(250);
    WaitNms(250);

    /* MMA1220、MMA1260 自测试*/
    AccTest(MMA1220);
    WaitNms(250);

    AccTest(MMA1260);
    WaitNms(250);

    for(;;)
    {
        u8NewAdcValue = AD1_GetValue16(gu8AccSelect);
        bNewTopValue = AccDataFormat(u8NewAdcValue);

        /* LCD 第一行显示最新的加速度值 */
        Lcd_Disp_Str(0x00, 0x05, gau8LcdFirstLine);

        /*是否有新的最大值出现？ */
        if(bNewTopValue)
{
            /*将 LCD 第一行的显示数据复制到第 2 行显示*/
            au8Lcd2ndLine[5] = gau8LcdFirstLine[9];
            au8Lcd2ndLine[6] = gau8LcdFirstLine[10];
            au8Lcd2ndLine[8] = gau8LcdFirstLine[12];

            /*  如果新加速度值超过加速度计的指标范围，显示"!!!" 报警标志*/
            if(gbOveraccState)
{
                au8Lcd2ndLine[12] = '!';
                au8Lcd2ndLine[13] = '!';
                au8Lcd2ndLine[14] = '!';
            }
            else
            {
                au8Lcd2ndLine[12] = '-';
                au8Lcd2ndLine[13] = '-';
                au8Lcd2ndLine[14] = '-';
            }
```

```
            /* 刷新 LCD 第 2 行的加速度值*/
            Lcd_Disp_Str(0x01, 0x05, au8Lcd2ndLine);
        }
    }
}
```

## 13.5 实例总结

本章通过利用 MC9S08JM60 单片机内部的 A/D 模块，辅之以加速度传感器 MMA1220、MMA1260，设计出了加速度传感器应用系统。其中显示部分采用 LCD，将加速度传感器测量的结果显示出来；单片机采用的是 Freescale 公司高性能的 MC9S08JM60 单片机，加速度传感器选用 Freescale 公司的线性加速度传感器。加速度传感器为一种新的传感领域，应用越来越广泛，因此本章的内容很有代表性。

# 第 14 章

# AD 模块：烟雾检测系统设计实例

随着人们对居住环境防火和安全方面的重视，烟雾检测系统得到了越来越高的重视。当有火情发生时，烟雾检测系统能以最快的速度检测报警，最大限度地减少灾害造成的人员伤亡和财产损失。本章将介绍基于 MC9S08JM60 单片机的烟雾检测系统的实现方法。

## 14.1 实例内容说明

烟雾检测系统利用光电传感器将光信号转换为电信号，这个电信号通过调理电路后，送入单片机的 A/D 接口，单片机通过检测这个电信号来判断是否有烟雾发生。如图 14-1 所示为系统的原理框图。

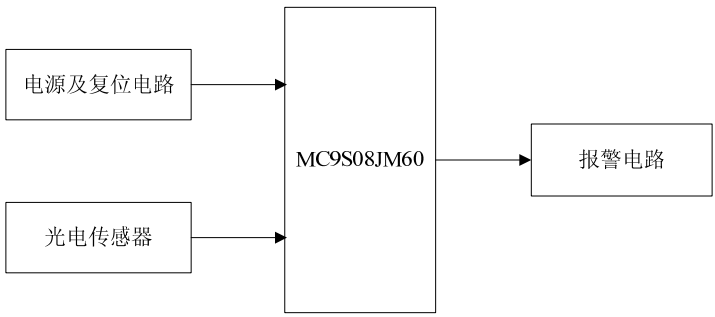



图 14-1 系统原理框图

由图 14-1 可以看出，系统的硬件电路比较简单，由光电传感器将光信号转换成电压信号，经调理电路，送入 MC9S08JM60 单片机的 AD 接口，经 AD 转换处理后，若有烟雾发生，则报警电路工作。

## 14.2 设计思路分析

烟雾检测系统的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、$I^2C$ 控制器、定时器等多种外设，方便进行系统设计。

分析图 14-1 所确定的系统原理框图，光电传感器是整个系统的关键，因此首先需选择合适的光电传感器。

光电传感器是通过把光强度的变化转换成电信号的变化来实现控制的。光电传感器一般由 3 部分组成：发送器、接收器和检测电路。

发送器对准目标发射光束，发射的光束来源于半导体光源，如发光二极管、激光二极管及红外发射二极管。接收器有光电二极管、光电三极管和光电池组成。在接收器后面为检测电路，它能滤出有效信号和应用该信号。

在本例中，我们选用红外发射二极管、红外接收器和检测电路设计一个光电传感器电路，实现对烟雾的检测。

## 14.3 硬件电路设计

烟雾检测系统设计实例主要由电源电路、报警电路、光电传感器电路和单片机电路组成。在本系统中，要求的电源电压为 3.3V，由于 3.3V 电源电路在第 5 章已经作了介绍，因此本章不再介绍，读者可以参看第 5 章的电源电路图。下面分别介绍光电传感器电路、报警电路和单片机电路。

### 14.3.1 光电传感器电路

光电传感器的电路如图 14-2 所示。光电传感器 3N261 的输出经运放 TLV2780 放大后送到 MC9S08JM60 的 PTB0 管脚，MC9S08JM60 的 PTB1 管脚控制光电传感器 3N261 工作与否，MC9S08JM60 的 PTB2 管脚控制运放 TLV2780 正常工作或关断。

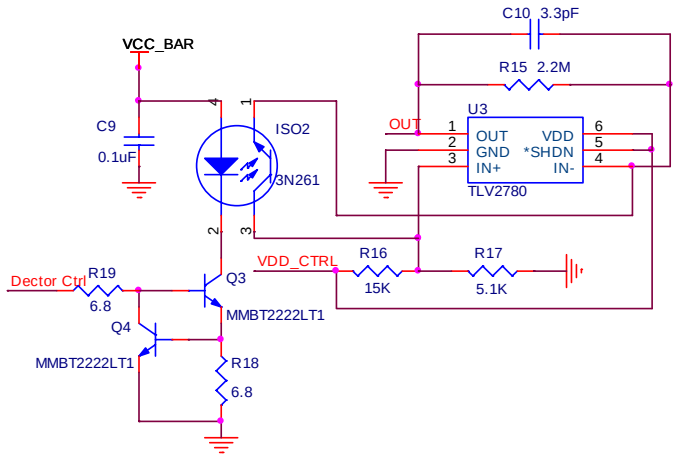



图 14-2　光电传感器电路图

## 14.3.2　报警电路

报警电路采用高分贝蜂鸣器，输入电压最大为 30V，烟雾检测系统的工作电压为 3V，所以在此采用 DC 升压转换芯片 TPS61040 来实现所需的电压。如图 14-3 所示为报警电路图。

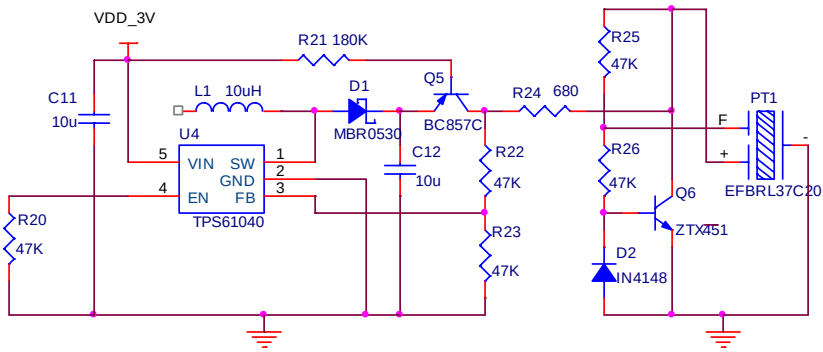



图 14-3　报警电路图

## 14.3.3　单片机电路

单片机的电路比较简单，只需要单片机的 PTB0～PTB3 与光电传感器电路、报警电路接口。另外单片机还需相应的振荡电路才能进行工作。如图 14-4 所示为具体的单片机电路图。

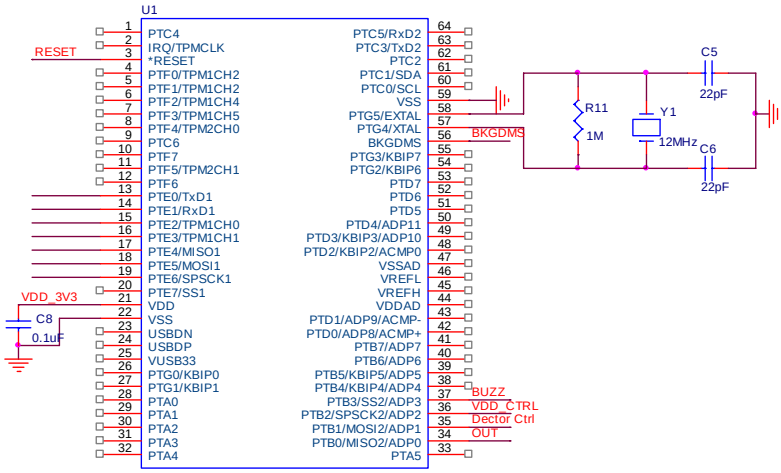



图 14-4　单片机电路图

# 14.4　软件设计与代码

## 14.4.1　软件设计流程

烟雾检测系统的软件主要由 3 部分组成：AD 采样程序、定时程序和主程序。其中 AD 采样程序是关键，它采样光电传感器的输出，将采样结果送给单片机作进一步的处理，单片机根据处理的结果执行相应的功能。如图 14-5 所示为系统的主程序流程图。

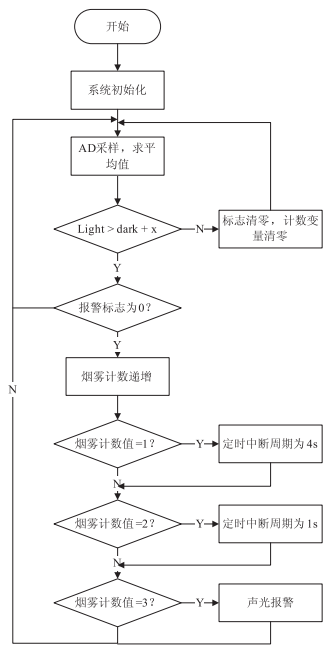



图 14-5　烟雾检测系统程序流程图

如图 14-5 所示，系统初始化完成后，即进入循环，启动 AD 采样，并处理 AD 采样的结果，单片机根据采样数据处理的结果，判断烟雾是否超标，从而启动声光报警。

### 14.4.2 AD 采样子程序

AD 采样子程序负责采样光电传感器输出的模拟量。主要包括 AD 初始化子程序、AD 转换子程序等。下面介绍具体的程序。

```
void LIGHT_AD_Init(void)
{
    setReg8(ADCSC1, 0x1F);              //关设备
    setReg8(ADCSC2, 0x00);              //控制寄存器 2 设置
    OutFlg = FALSE;
    ModeFlg = STOP;                     //设备状态设置
   setReg8(ADCCFG, 0x07);               //设置分频位
}

void Sample (void)
{
  unsigned char i, temp;

  temp = PTBD;                          //保存端口 B 的状态
  OAPOWER_SetVal();                     // Turn on OA.
  LED_SetVal();

  for(i =0;i< 4;i++)
  {
    LIGHT_AD_Measure(0x01);             //启动 A/D 转换
    LIGHT_AD_GetValue(dark_buffer);     //读取 A/D 转换结果
  }
  IRLED_SetVal();                       // Turn on IRLED

  for(i =0;i< 4;i++)
  {
    LIGHT_AD_Measure(0x01);
    LIGHT_AD_GetValue(light_buffer);
  }
  PTBD = temp;
}
```

### 14.4.3 定时程序

本例中使用定时器 0 的通道 0，工作在 PWM 模式下，周期可变，初始周期为 8s。定时程序主要包括定时器初始化子程序、定时周期设置子程序等。下面给出定时程序的具体

程序代码。

```
void Timer_Variable_Init(void)
{
   setReg8(TPM1SC, 0x40);            //关设备，开设备中断
   setReg8(TPM1C0SC, 0x3C);          //设置定时器为 PWM 模式，输出电平为低
   RatioStore = 0xFFFF;              //存储比率的初始值
  SetPV((byte)0x03);                 //设置定时器预分频计数器
  setReg16(TPM1C0V, 0xB71BU);
  setReg16(TPM1MOD, 0xB71AU);        //设置模式寄存器
  setReg8Bits(TPM1SC, 0x50);         //启动定时器
}

byte Timer_Variable_SetDutyMS(word Time)
{
  dlong rtval;
  if (Time > 0x1F40) {               //参数超范围?
    return ERR_RANGE;                //返回出错标志
  }
  PE_Timer_LngMul((dword)Time, 0x083126E9LU, &rtval); //调用内部函数
  if (PE_Timer_LngHi3(rtval[0], rtval[1], &RatioStore))//结果大于 65536?
  {
     RatioStore = 0xFFFF;            //使用最大可取值
  }
  SetRatio();                        //设置比较和模式寄存器
 return ERR_OK;
}
```

### 14.4.4 主程序

系统经过一系列初始化后，即进入主程序工作。主程序调用 AD 采样程序、定时子程序等来检测烟雾，其中 AD 采样子程序采用查询方式，定时器工作在 PWM 模式下。下面给出烟雾检测系统的主程序。

```
void main(void)
{
    PE_low_level_init();

    for(;;)
{
    Sample();
    Average();

    if (light_average > (dark_average + 30))
    {
```

```
        if (alarm_flag == 0)
        {
          smoke_detect_count++;            // 烟雾计数值递增
          smoke_detect_count &= 0x03;      // 烟雾计数值最大为 3

          if (smoke_detect_count == 1)
          {
             Timer_Variable_SetDutyMS(4000);  //定时周期为 4s
          }
          else if (smoke_detect_count == 2)
          {
            Timer_Variable_SetDutyMS(1000);   //定时周期为 1s
          }
          else if (smoke_detect_count == 3)
          {
            alarm_flag = 1;                   // 报警标志置 1
            LED_SetVal();                     // 点亮 LED
          }
        }
      }
      else
      {
        LED_ClrVal();
        alarm_flag = 0;                      // 没有烟雾出现，则标志清零
        smoke_detect_count = 0;              // 烟雾计数值清零
        Timer_Variable_SetDutyMS(8000);
      }
    }
  }
```

## 14.5 实例总结

本章通过利用单片机内部的 AD 模块，辅之以光电传感器电路、声光报警电路等外围电路，设计出了简单的烟雾报警系统。本章重点采用了光电传感器电路，并穿插介绍了光电传感器的背景知识。虽然本章实现的烟雾检测系统相对简单，但是读者可以在本章的基础上完善相应的功能，实现更加复杂的系统。如增加 LED 显示，将烟雾的具体值显示出来；增加按键功能，实现烟雾阈值设定等。

# 第 15 章

# PWM 模块:寻迹智能小车控制系统设计实例

随着社会的发展，各种智能化的产品日益普及。为了实现产品的便携性、低成本及对电源的限制，小型直流电机应用相当广泛。对直流电机的速度调节，主流的技术就是采用 PWM 技术。本章将介绍基于 MC9S08JM60 单片机的 PWM 模块实现的寻迹智能小车控制系统的开发。

## 15.1 实例内容说明

寻迹智能小车的具体功能为：按照一定的路线到达指定的位置，完成一定的动作后再走到下一个目标位置。智能小车的行走机构为车轮方式：左右轮为驱动轮，前后为支撑轮，由此构成四轮结构。左右轮除负责前进后退外，当两轮的转速不同时，还可以实现曲线行走和原地旋转。如图 15-1 所示为系统的原理框图。

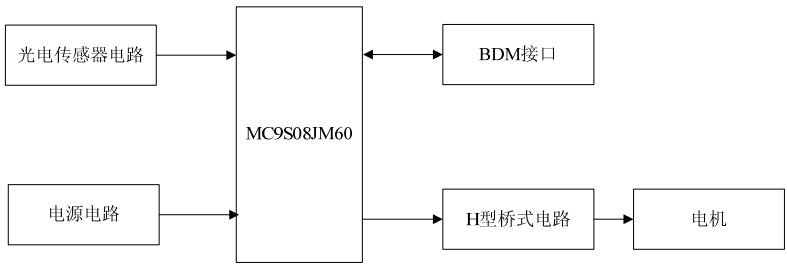



图 15-1 系统原理框图

由图 15-1 可以看出，由光电传感器电路检测智能小车的位置状态，通过 CPU 判断驱

动 H 形桥式电路控制电机工作，使小车按照预定方案沿着轨迹移动。下面对具体的设计思路进行介绍。

## 15.2 设计思路分析

寻迹小车的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、I2C 控制器、定时器等多种外设，方便进行系统设计。

分析图 15-1 所确定的系统原理框图，可知 PWM 是实现整个系统的关键，因此首先需要了解 PWM 的基本原理及其实现方法。

### 15.2.1 PWM 基本原理

PWM 是通过控制固定电压的直流电源开关频率，从而改变负载两端的电压，进而达到控制要求的一种电压调整方法。PWM 可以应用在许多方面，如电机调速、温度控制、压力控制等。

在 PWM 驱动控制的调整系统中，按一个固定的频率来接通和断开电源，并根据需要改变一个周期内“接通”和“断开”时间的长短。通过改变直流电机电枢上电压的“占空比”来改变平均电压的大小，从而控制电动机的转速。因此，PWM 又被称为“开关驱动装置”。

如图 15-2 所示，在脉冲作用下，当电机通电时，速度增加；电机断电时，速度逐渐减小。只要按一定规律，改变通、断电的时间，即可让电机转速得到控制。

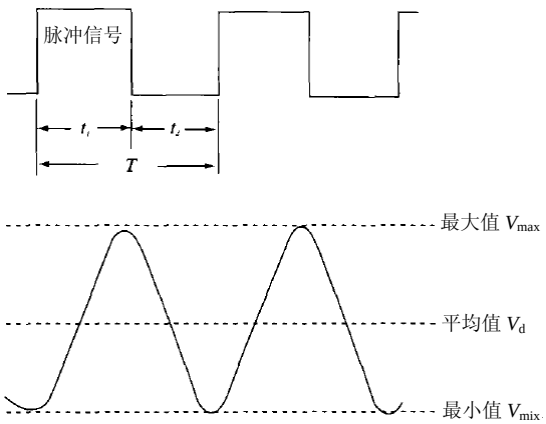



图 15-2 电枢电压“占空比”与平均电压的关系图

设电机始终接通电源时，电机转速最大为 $V_{max}$，设占空比为 $D=t_1/T$，则电机的平均速度为：

$$V_d=V_{max}\times D \qquad (1)$$

式中，$V_d$—电机的平均速度；

$V_{max}$—电机全通电时的速度（最大）；

$D=t_1/T$—占空比。

由公式（1）可见，当我们改变占空比 $D=t_1/T$ 时，就可以得到不同的电机平均速度 $V_d$，从而达到调速的目的。严格地讲，平均速度 $V_d$ 与占空比 $D$ 并不是严格的线性关系，在一般的应用中，可以将其近似地看成线性关系。

### 15.2.2 实现方法

PWM 信号的产生通常有两种方法：一种是软件的方法；另一种是硬件的方法。硬件方法的实现已有很多文章介绍，这里不作赘述。本文主要介绍利用 MC9S08JM60 单片机对 PWM 信号的软件实现方法。

HCS08 系列单片机基本都有定时器模块 TPM/PWM，通过控制定时器初值，从而可以实现从定时器的输出口输出不同占空比的脉冲波形。

## 15.3 硬件电路设计

寻迹小车设计实例主要由 H 形桥式电路、光电传感器电路、电源电路和单片机电路组成。在本系统中，单片机要求的电源电压为 3.3V，由于 3.3V 电源电路在第 5 章已经作了介绍，因此本章不再介绍，读者可以参看第 5 章的电源电路图。H 桥电路所需电源电压为 12V。下面分别介绍 H 形桥式电路、光电传感器电路、电源电路和单片机电路。

### 15.3.1 H 形桥式电路

智能车整体的运行性能主要取决于电机驱动系统。小车的驱动不但要求电机驱动系统具有高转矩重量比、宽调速范围、高可靠性，而且电机的转矩—转速特性受电源功率的影响，这就要求驱动具有尽可能宽的高效率区。我们所使用的电机一般为直流电机，直流电机驱动电路使用最广泛的就是 H 形全桥式电路。这种驱动电路可以很方便地实现直流电机的四象限运行，分别对应正转、正转制动、反转、反转制动。本例中实现的 H 形全桥电路如图 15-3 所示。

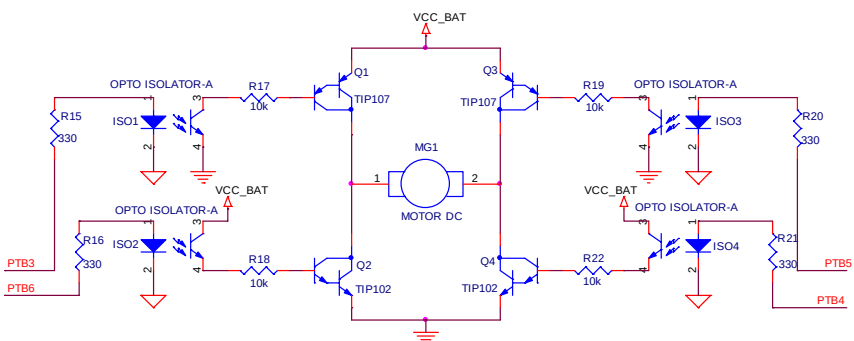



图 15-3 H 形全桥式电路图

如图 15-3 所示，全桥式驱动电路的 4 只开关管都工作在斩波状态，Q 1、Q4 为一组，Q2、Q3 为另一组，两组的状态互补，一组导通则另一组必须关断。当 Q 1、Q4 导通时，Q2、Q3 关断，电机两端加正向电压，可以实现电机的正转或反转制动；当 Q2、Q3 导通时，Q 1、Q4 关断，电机两端为反向电压，电机反转或正转制动。

在小车动作的过程中，我们要不断地使电机在四个象限之间切换，即在正转和反转之间切换，也就是在 Q 1、Q4 导通且 Q2、Q3 关断，到 Q 1、Q4 关断且 Q2、Q3 导通，这两种状态之间转换。在这种情况下，理论上要求两组控制信号完全互补，但是，由于实际的开关器件都存在开通和关断时间，绝对的互补控制逻辑必然导致上下桥臂直通短路，例如在上桥臂关断的过程中，下桥臂导通了。

因此，为了避免直通短路且保证各个开关管动作之间的协同性和同步性，两组控制信号在理论上要求互为倒相的逻辑关系，而实际上却必须相差一个足够的死区时间。

驱动电流不仅可以通过主开关管流通，而且还可以通过续流二极管流通。当电机处于制动状态时，电机便工作在发电状态，转子电流必须通过续流二极管流通，否则电机就会发热，严重时烧毁。

开关管的选择对驱动电路的影响很大，开关管的选择宜遵循以下原则：

①由于驱动电路是功率输出，要求开关管输出功率较大；

②开关管的开通和关断时间应尽可能小；

③小车使用的电源电压不高，因此开关管的饱和压降应该尽量低。

在此，我们选用大功率达林顿管 TIP102 和 TIP107。最终的电机驱动 H 形全桥式电路如图 15-3 所示。

H 形半桥电路用来控制伺服电机，实现小车的前进。H 形半桥式电路的工作原理和 H 形全桥式电路的工作原理基本相同，但 H 形半桥式电路使用一组开关，使用一个控制信号来控制该组开关导通，从而驱动电机工作。在此我们采用 MC9S08JM60 的 PTE2 管脚输出的 PWM 信号来控制前进电机。本例实现的 H 形半桥电路如图 15-4 所示。

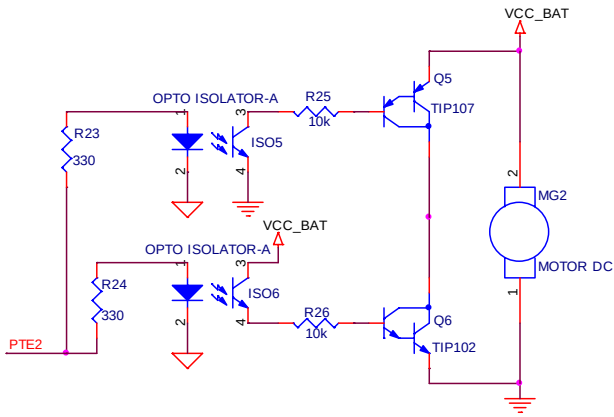



图 15-4　H 形半桥式电路图

### 15.3.2 光电传感器电路

光电传感器电路如图 15-5 所示。图中采用的红外光电检测管为单光束反射取样式光电传感器 QRB1114。它由红外发射器和感光器组成。通常这两部分是封装在一起的。

QRB1114 正常工作时，红外线发光二极管发出红外线。如果检测面表面光洁度平整，发出的红外线经检测面反射后在光敏三极管产生光电效应。当在白线区时，发射的红外线大部分被反射到光敏三极管；而在黑线区，发射的红外线大部分被吸收，少部分被反射到光敏三极管，所以通过检测光敏三极管的输出电压高低来判断小车有无偏离预定路线。图 15-5 中 330Ω 的电阻主要是限流作用，15kΩ 的下拉电阻将光敏三极管的输出电流转换成输出电压供单片机来处理。

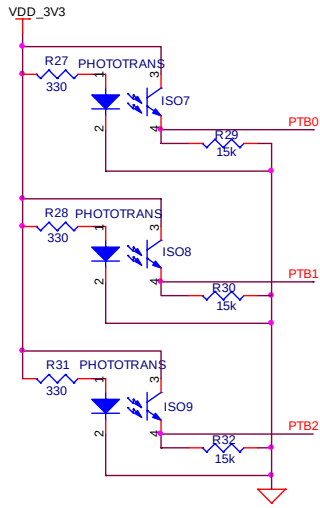



图 15-5 光电传感器电路图

### 15.3.3 电源电路

寻迹智能车中，系统要求的电源电压主要有 12V 和 3.3V。12V 为 H 形桥式电路的工作电源，由电池来供给。图 15-6 给出 12V 转 5V 的电源电路图，5V 输出作为 3.3V 电源电路的输入。单片机要求的电源电压为 3.3V，由于 3.3V 电源电路在第 5 章已经作了介绍，因此本章不再介绍，读者可以参看第 5 章的电源电路图。

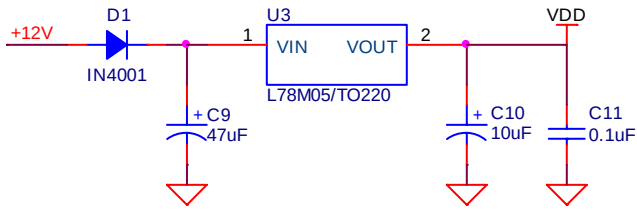



图 15-6　12V 转 5V 的电源电路图

### 15.3.4 单片机电路

单片机的电路比较简单，只需要单片机的 PTB 口与 H 形桥式电路、光电传感器电路接口、PTE2 管脚与 H 形半桥电路接口。另外还需相应的振荡电路才能使单片机进行工作，如图 15-7 所示为具体的单片机电路图。

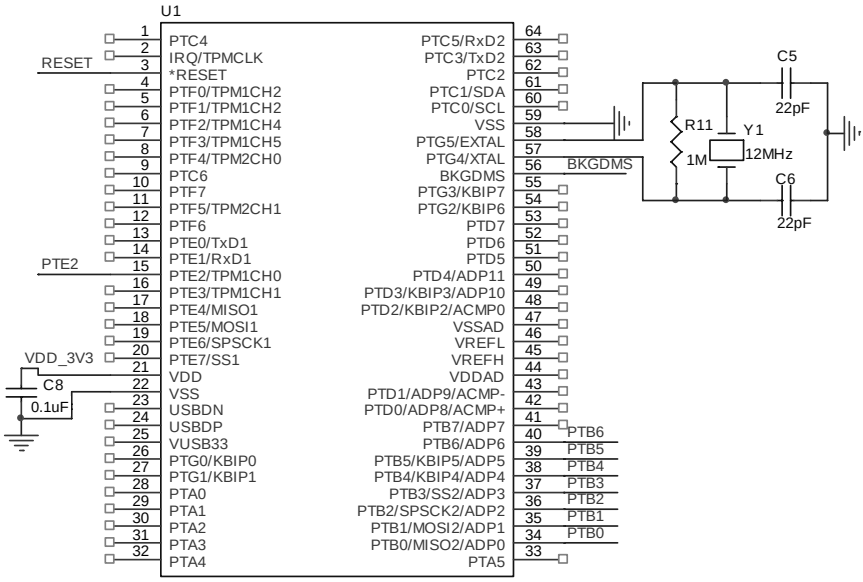



图 15-7 单片机电路图

# 15.4 软件设计与代码

## 15.4.1 软件设计流程

寻迹智能车系统程序主要由主程序、传感器采样处理子程序、智能车移动子程序等组成，其中传感器采样处理子程序是关键。传感器采样处理子程序通过采样传感器的输出，经过 A/D 转换后，在单片机内进行数据处理，以获取智能车的状态。传感器采样处理子程序基于状态机设计，其程序流程图如图 15-8 所示。

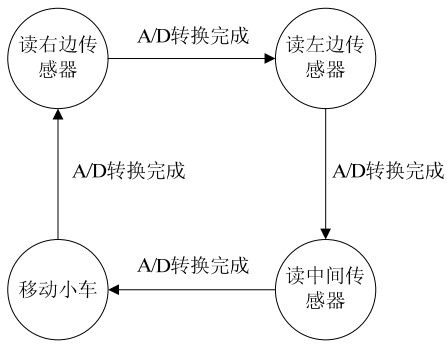



图 15-8 传感器处理子程序流程图

下面分别介绍具体的程序设计。

### 15.4.2 传感器采样处理子程序

传感器采样处理子程序基于状态机设计，共有 4 个状态，程序检测到系统处于哪个状态，则执行相应状态下的程序。下面介绍具体的程序。

```
void sensor_check(void)
{
    switch (sensor_state)
    {
      case move:
       move_car();
       sensor_timer = 0;
       sensor_state = right;
      break;

      case right:
      if (sensor_timer >= 2)
      {
          Sensor_MeasureChan(0, 0x01) ;
          Sensor_GetChanValue16(0x01, &rightRd);
          sensor_state = left;
      }
      else
      {
         sensor_state = right;
      }
      break;

      case left:
      if (sensor_timer >= 4)
      {
          Sensor_MeasureChan(0, 0x00) ;
          Sensor_GetChanValue16(0x00, &leftRd);

          sensor_state = middle;
      }
      else
      {
         sensor_state = left;
      }
      break;

      case middle:
      if (sensor_timer >= 6)
```

```
        {
            Sensor_MeasureChan(0, 0x02) ;
            Sensor_GetChanValue16(0x02, &midRd);

            sensor_state = move;
        }
        else
        {
           sensor_state = middle;
        }
        break;
      }
  }
```

### 15.4.3 智能车移动子程序

传感器采样处理子程序执行后，系统将根据采样处理的结果控制智能车的移动。智能车移动子程序设计也是基于状态机的，共有停止、左转、向前、右转 4 个状态，当条件满足时，则从一个状态转换到另一个状态。下面给出智能车移动子程序代码。

```
void move_car(void)
{
    switch (car_state)
    {
       case stop:
         if (midRd>80 && leftRd>80 && rightRd>80)//车未在线上?
         {
             PWM_5mS_Disable();
             Turning_PutVal(0x00);//停止驱动电机
         }
         else
         {
               PWM_5mS_Enable();  //使能 PWM 模式
               car_state = straight;
          }
          break;

        case straight:
         if (midRd>80 && leftRd>80 && rightRd>80)
         {
             PWM_5mS_Disable();
             Turning_PutVal(0x00); //停止驱动电机

             car_state = stop;
         }
```

```
  else if (leftRd<40 && rightRd>80)
   {
       Turning_PutBit(2, 0);  /关闭其他晶体管
       Turning_PutBit(3, 0);

       Turning_PutBit(0, 1);  //驱动电机上电，左转
       Turning_PutBit(1, 1);

       car_state =left_turn;
       turn = 1;
   }
  else if (rightRd<40 && leftRd>80)
  {
       Turning_PutBit(0, 0);  //关闭其他晶体管
       Turning_PutBit(1, 0);

       Turning_PutBit(3, 1);   //驱动电机上电，右转
       Turning_PutBit(2, 1);
       car_state = right_turn;
       turn = 1;
 }
break;

case left_turn:
  if (turn_timer >= 1500)
  {
     if (midRd>80 && leftRd>80 && rightRd>80)  //车未在线路上?
     {
         PWM_5mS_Disable();     //关闭所有电机
         Turning_PutVal(0x00);
         car_state = stop;
     }
     turn_timer = 0;
  }

 if (midRd>80 && leftRd<40 && rightRd>80)//中间传感器未在线路上?
  {
       Turning_PutBit(0, 1);      //电机上电，左偏
       Turning_PutBit(1, 1);
       Turning_PutBit(2, 0);
       Turning_PutBit(3, 0);
  }
  else if (midRd<40 && leftRd>80 && rightRd>80)
  {
       Turning_PutBit(2, 0);        //左边制动
       Turning_PutBit(1, 0);
```

```
        Turning_PutBit(0, 1);
        Turning_PutBit(3, 1);
        car_state = stop;
    }
    else if (rightRd<40 && leftRd>80)  //车往左边大偏?
    {
        Turning_PutBit(2, 0);          //左边制动
        Turning_PutBit(1, 0);
        Turning_PutBit(0, 1);
        Turning_PutBit(3, 1);

       car_state = right_turn;
    }
    break;

case right_turn:
    if (turn_timer >= 1500)
    {
        if (midRd>80 && leftRd>80 && rightRd>80)
        {
            PWM_5mS_Disable();          //关闭所有电机
            Turning_PutVal(0x00);

            car_state = stop;
        }
        turn_timer = 0;
   }

if (midRd>80 && rightRd<40 && leftRd>80) //中间传感器未在线路上?
    {
        Turning_PutBit(3, 1);           //电机上电，右转
        Turning_PutBit(2, 1);
        Turning_PutBit(1, 0);
        Turning_PutBit(0, 0);
    }
    else if (midRd<40 && leftRd>80 && rightRd>80)//小车已右转?
    {
       Turning_PutBit(2, 0);            //右边制动?
       Turning_PutBit(1, 0);
       Turning_PutBit(0, 1);
       Turning_PutBit(3, 1);

       car_state = stop;
    }
    else if (leftRd<40 && rightRd>80)   //小车右大偏?
    {
```

```
                Turning_PutBit(2, 0);           //右边制动
                Turning_PutBit(1, 0);
                Turning_PutBit(0, 1);
                Turning_PutBit(3, 1);

                car_state = left_turn;
            }
            break;
        }
    }
```

### 15.4.4 主程序

寻迹智能车系统的主程序比较简单，系统经过初始化后，即进入主程序工作。主程序调用传感器采样处理子程序，并根据采样处理程序的结果控制小车移动，实现小车的寻迹功能。下面给出寻迹智能车的主程序。

```
void main(void)
{
   PE_low_level_init();         //系统初始化

ensor_timer = 0;             //系统变量初始化
   car_state = stop;
   sensor_state = right;
   go_timer = 0;

   Turning_PutVal(0x00);        //关闭电机

   for(;;)
   {
      sensor_check();          //传感器采样处理子程序
   }
 }
```

## 15.5 实例总结

本章通过利用单片机内部的 PWM 模块，辅之以 H 桥电路、光电检测电路等外围电路，设计出了寻迹智能小车。重点从系统功能说明出发，对系统的原理框图、PWM 的原理和实现方法进行了分析。在硬件设计部分给出了较为详细的硬件电路设计图，并对设计原理进行了介绍。本章介绍的寻迹智能小车采用红外光电传感器来检测智能小车的运动状态，通过改变传感器的类型，如超声波传感器，智能小车就可以判断周围障碍物的距离，从而改变运动方向，实现避障功能。

# 第 16 章

# 通用串口通信模块：单片机与 PC 通信设计实例

单片机在控制本地的外围器件时信息的交互是通过 8 位并行数据线进行的，但是在较大规模的现代控制系统中，单片机还需要控制远端的设备，需要实现单片机的通信。使用单片机自带的串行通信模块，可以方便地实现单片机与 PC 的串行通信。本章将介绍基于 MC9S08JM60 单片机的 PC 通信设计方法。

## 16.1 实例内容说明

单片机与 PC 通信一般采用串口来实现。Freescale 单片机基本都有串口模块，因此实现起来非常容易。由于单片机的电平与 PC 串口的电平不一致，因此需要电平转换芯片实现接口转换。如图 16-1 所示为系统的原理框图。

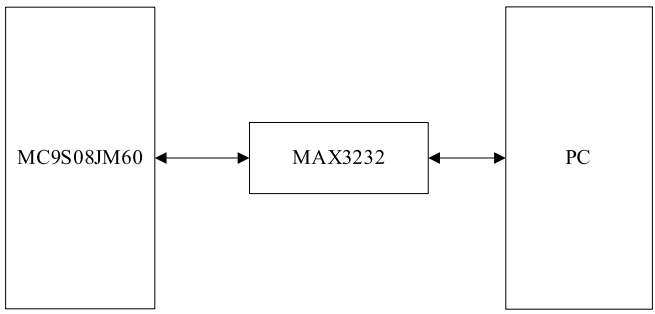



图 16-1 系统原理框图

## 16.2 设计思路分析

本系统的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、$I^2C$ 控制器、定时器等多种外设，方便进行系统设计。

按照图 16-1 所确定的系统原理框图，电平转换芯片是整个系统的关键，因此首先需选择合适的电平转换接口芯片。

### 16.2.1 电平转换芯片

PC 的串口是 RS-232C 标准串口。RS-232C 标准对电器特性、逻辑电平和各种信号线功能都做了规定。对于数据，逻辑“0”的电平高于-3V，逻辑“1”的电平低于+3V；对于控制信号，接通状态（ON）即信号有效的电平高于+3V，断开状态（OFF）即信号无效的电平低于-3V。也就是当传输电平的绝对值大于 3V 时，电路可以有效地检查出来。介于-3V～+3V 之间的电压无意义，低于-15V 或高于+15V 的电压也认为无意义。因此，在实际工作时，应保证电平在±（3～15）V 之间。RS-232C 标准用正负电压表示逻辑状态，与单片机中 TTL 以高低电平表示逻辑状态的规定不同。因此，为了能够实现 PC 与单片机的串行通信，必须在 RS-232C 与 TTL 电路之间进行电平和逻辑关系的转换。

能够实现 RS-232C 电平与 TTL 电平转换的专用芯片有很多种。本例选用常用的 Maxim 公司的 MAX3232 集成芯片。MAX3232 是一款可以实现 EIA-232 接口的低功耗电平转换芯片，它包含两路收发器，数据传输速率可保证为 250kbps，具有强静电保护能力。

## 16.3 硬件电路设计

本系统的硬件接口电路比较简单，主要就是单片机与 MAX3232 芯片的连接。为了便于读者理解接口，在介绍接口电路前，首先介绍 MAX3232 芯片。

### 16.3.1 MAX3232 芯片介绍

MAX3232 是一款可以实现 EIA-232 接口的低功耗电平转换芯片，该芯片具有以下特性：

- 宽电压供电。供电电压：0.3V～6V。
- 250kbps 数据传输率。
- 低功耗的电流为 1uA。
- 增强性 ESD 规范。

为了便于进行硬件电路的设计，下面给出该芯片的管脚图，如图 16-2 所示。

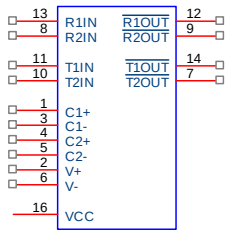



图 16-2　MAX3232 的管脚图

由图 16-2 可以看出，该芯片只有 16 个管脚。下面对具体的管脚进行介绍：

- C1+、C1-：电压增倍的充电电容的正极和负极。
- C2+、C2-：倒置充电电容的正极和负极。
- V+、V-：由充电电容产生的 5.5V 的正极和负极。
- VCC：电源管脚。
- GND：接地管脚。
- R1IN、R2IN：RS232 的输入。
- R1OUT、R2OUT：TTL/CMOS 的输出。
- T1IN、T2IN：TTL/CMOS 的输入。
- T1OUT、T2OUT：RS232 驱动的输出。

经过对 MAX3232 芯片的介绍，读者应该对 MAX3232 有了基本的认识，下面介绍其他电路的具体设计。

## 16.3.2　串口电路

硬件的接口电路相对简单，主要就是串口的设计。Freescale HCS08 单片机基本都有串口模块，因此 Freescale HCS08 单片机很容易通过片内的串口实现与 MAX3232 芯片的接口。如图 16-3 所示为接口电路图。

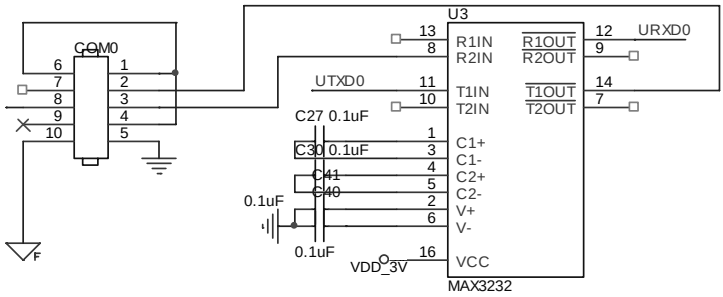



图 16-3　接口电路图

## 16.3.3　单片机电路

单片机的电路比较简单，如图 16-4 所示为具体的单片机电路图。

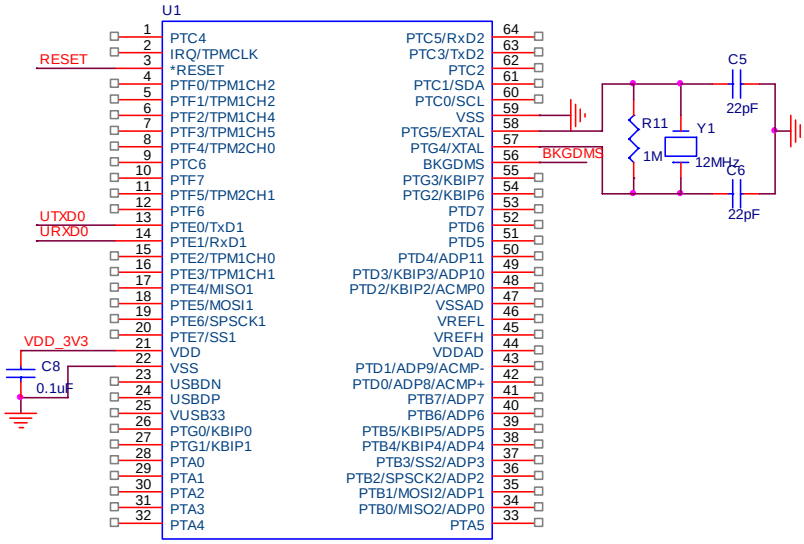



图 16-4　单片机电路图

# 16.4　软件设计与代码

## 16.4.1　软件设计流程

单片机与 PC 通信设计实例的系统软件比较简单，实现的主要功能为：单片机接收 PC 发送过来的数据，并将数据重新发回 PC。实例的软件流程图如图 16-5 所示。

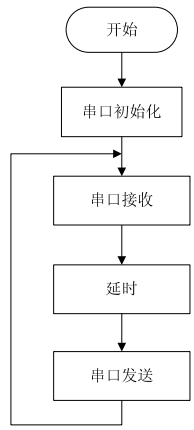



图 16-5　主程序软件流程图

下面分别介绍具体的程序。

### 16.4.2 串口初始化程序

串口初始化主要负责完成串行口数据位、波特率、中断设置等。下面介绍具体的程序。

```
void Uart_Init(void)
{
  SerFlag = 0;                        //标志位清零
  Uart_InpLen = 0;                    //接收缓冲器数据个数清零
  InpIndxR = InpIndxW = 0;
  Uart_OutLen = 0;                    //发送缓冲器数据个数清零
  OutIndxR = OutIndxW = 0;

  setReg8(SCI1C1, 0x00);              //设置串行数据位格式
  setReg8(SCI1C3, 0x00);              //禁止错误中断
  setReg8(SCI1S2, 0x00);
  setReg8(SCI1C2, 0x00);              //禁止所有中断
  SCI1BDH = 0x00;                     //波特率寄存器设置
  SCI1BDL = 0x4E;
  SCI1C3 |= 0x0F;                     //使能错误中断
  /* 使能发送器、接收器，使能接收中断 */
  SCI1C2 |= ( SCI1C2_TE_MASK | SCI1C2_RE_MASK | SCI1C2_RIE_MASK);
}
```

### 16.4.3 串口中断服务程序

串口中断包括串口接收中断、串口发送中断和错误中断，所以串口中断服务程序也有串口接收中断服务程序、串口发送中断服务程序、串口错误中断服务程序。下面给出串口中断服务程序的子程序。

```
ISR(Uart_InterruptRx)
{
  byte StatReg = SCI1S1;              //状态标志的临时变量
  Uart_TComData Data;                 //存储数据的临时变量
  byte OnFlags = 0;                   //用于标志的临时变量

  Data = SCI1D;                       //从接收器中读取数据
  if(Uart_InpLen < Uart_INP_BUF_SIZE) //缓冲器中的数据未满
  {
    Uart_InpLen++;
    InpBuffer[InpIndxW] = Data;       //将接收到的数据存入接收缓冲器中
    InpIndxW = (byte)((InpIndxW+1) & (Uart_INP_BUF_SIZE-1)); //更新索引
    OnFlags |= ON_RX_CHAR;            //置标志位 OnRXChar
    if(Uart_InpLen== Uart_INP_BUF_SIZE) //缓冲器满?
{
     OnFlags |= ON_FULL_RX;           //置标志位 OnFullRxBuff
```

```
    }
  }
  else
  {
    SerFlag |= FULL_RX;                //缓冲器溢出
    OnFlags |= ON_ERROR;               //置标志位 OnError
  }
  if(OnFlags & ON_ERROR)               //错误标志位置 1?
  {
    Uart_OnError();                    //调用串口错误处理程序
  }
  else {
    if(OnFlags & ON_RX_CHAR)           //标志位 OnRxChar 置 1?
    {
      Uart_OnRxChar();
    }
    if(OnFlags & ON_FULL_RX)          //标志位 OnFullRxBuf 置 1?
    {
      Uart_OnFullRxBuf();
    }
  }
}

ISR(Uart_InterruptTx)
{
  byte OnFlags = 0;                    //标志位临时变量

  if(SerFlag & RUNINT_FROM_TX)         //标志位 RUNINT_FROM_TX 置 1?
  {
    OnFlags |= ON_TX_CHAR;
  }
  SerFlag &= ~RUNINT_FROM_TX;          //标志位 RUNINT_FROM_TX 清零
  if(Uart_OutLen) {                    //发送缓冲器还有数据?
    Uart_OutLen--;
    SerFlag |= RUNINT_FROM_TX;         //标志位 RUNINT_FROM_TX 置 1
    (void)SCI1S1;                      //复位中断请求标志
    SCI1D = OutBuffer[OutIndxR];       //发送数据
    OutIndxR = (byte)((OutIndxR+1) & (Uart_OUT_BUF_SIZE-1));
  }
  else
  {
    OnFlags |= ON_FREE_TX;             //标志位 OnFreeTxBuf 置 1
    SCI1C2_TIE = 0;                    //禁止发送中断
  }
```

```
  if(OnFlags & ON_TX_CHAR)          //标志位 OnTxChar 置 1?
  {
    Uart_OnTxChar();                //调用串口发送数据函数
  }
  if(OnFlags & ON_FREE_TX)          //标志位 OnFreeTxBuf 置 1?
  {
    Uart_OnFreeTxBuf();
  }
}
```

### 16.4.4 主程序

系统主程序比较简单。经过系统初始化和串口初始化后，即进入主程序工作。主程序调用按串口接收函数接收 PC 发送的数据，延时一段时间，调用串口发送函数将接收到的数据发送到 PC。下面给出具体的主程序。

```
void main(void)
{
   PE_low_level_init();

   for(;;)
   {
          Uart_RecvChar(&Receive_Data);
          Delay_Time(100);
          Uart_SendChar(Receive_Data);
    }

}
```

## 16.5 实例总结

本章设计了简单的单片机与 PC 通信的实例。单片机的串口通信采用中断方式接收 PC 发送的数据，发送数据也采用中断方式。本例较为简单地阐述了 MC9S08JM60 单片机串口通信的一般设计步骤。通过本章的学习，读者可了解单片机串口通信系统的设计方式与过程。

# 第 17 章

# RS-485 总线通信模块：现场监测系统设计实例

随着数字控制技术的发展，由单片机构成的控制系统日益复杂。在一些要求响应速度快、实时性强、控制量多的应用场合，单个单片机构成的系统往往难以胜任。这时，由多个单片机结合 PC 组成分布式系统是一个比较好的解决方案。在这样的分布式系统中，大都采用 RS-485 接口连接 PC 和单片机。

RS-485 总线在工业应用中具有十分重要的地位，大量的已有工业设备均提供 RS-485 接口。本章将介绍基于 MC9S08JM60 单片机的 485 总线现场监测系统，首先介绍硬件电路设计，然后再介绍软件设计。

## 17.1 实例内容说明

485 总线现场监测系统是一个以 PC 作为主控机、多个单片机作为从机构成的。单片机组成的各个节点负责采集终端设备的状态信息，主机以轮询的方式向各个节点获取这些设备信息，并根据信息内容进行相关操作。如图 17-1 所示为系统的原理框图。

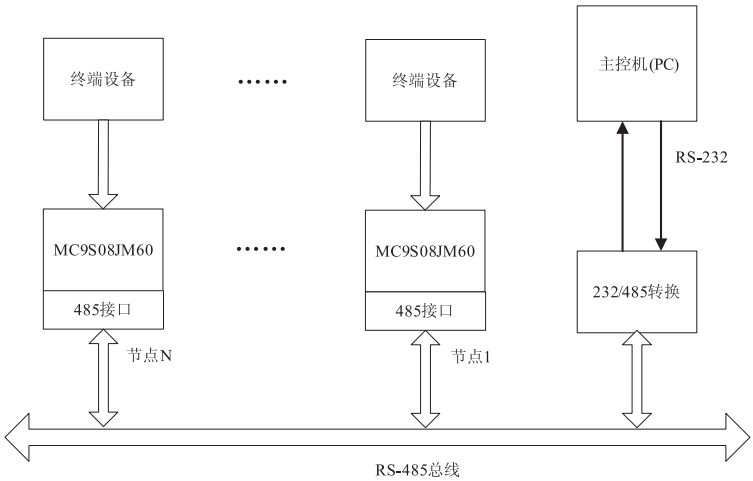



图 17-1　系统原理框图

由图 17-1 可以看出，整个系统分为 3 个部分。

（1）主控制部分

该部分由 PC 和 232/485 转接设备构成，它负责轮询各个节点以获取信息。由于 PC 不具备 485 接口，所以必须通过 232/485 转接设备接入 485 总线网络。

（2）数据采集部分

该部分是一个具有 485 接口的 MC9S08JM60 单片机系统，它位于各个终端设备处，响应主控机发送来的查询命令，并将被监测设备的状态信息回送给主控机。MC9S08JM60 的 I/O 口与被监测的终端设备相连，以获取所需的状态信息。

（3）终端设备

该部分是被监测的设备，一般为带有传感器的各种具体机器设备。

## 17.2　设计思路分析

现场监测系统的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、I2C 控制器、定时器等多种外设，方便进行系统设计。

分析图 17-1 所确定的系统原理框图可知，关键在于 485 总线的设计，因此首先需要了解 RS-485 通信标准的相关知识。

### 17.2.1　RS-485 总线

RS-485 是 RS-422A 的变形。RS-422A 为全双工工作方式，可以同时发送和接收数据，而 RS-485 则为半双工工作方式，在某一个时刻，一个发送另一个接收。RS-485 是一种多

发送器的电路标准，它扩展了 RS-422A 的性能。在同一个 RS-485 网络中，可以有多达 32 个模块，这些模块可以是被动发送器、接收器或收发器。RS-485 电路允许公用电话线通信。电路结构式平衡，连接电缆两端有终端电阻，在平衡电缆上挂发送器、接收器或收发器。RS-485 标准没有规定何时控制发送器发送和接收器接收数据的规则，其电缆选择比 RS-422A 更为严格。

### 17.2.2 RS-485 驱动芯片的选取

本例最重要的器件是 RS-485 驱动芯片，它决定了整个 485 总线网络的建立。单片机芯片选取 Freescale 公司的 MC9S08JM60，数据采集不是本章的重点，不作详细介绍。

TI 公司的 SN75176 芯片是一款用于 RS-485 通信的低功耗半双工收发器件，芯片内部集成了一个驱动器和一个接收器，符合 RS-422A 和 RS-485 通信标准。SN75176 的性能和特点符合本例的需要，性价比较高，所以选用它来作为本例的 485 驱动芯片。

## 17.3 硬件电路设计

本系统的硬件电路主要包括两部分：主控机（PC）的 485 通信接口及带有 485 接口的单片机系统。在介绍具体的硬件电路前，先介绍一下 SN75176 芯片。

### 17.3.1 SN75176 芯片介绍

MAX3232 是一款可以实现 EIA-232 接口的低功耗电平转换芯片，该芯片具有以下特性：

- +5V 单电源工作。
- 高驱动能力，最大可达 60mA。
- 驱动器具有过载保护功能。
- 具有接收/驱动使能端。

为了便于进行硬件电路的设计，下面给出该芯片的管脚图，如图 17-2 所示。

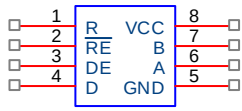



图 17-2 SN75176 的管脚图

由图 17-2 可以看出，该芯片只有 8 个管脚。下面对具体的管脚进行介绍。

- R：接收器输出。
- RE：接收器输出使能。引脚为 0，允许接收器输出；引脚为 1，禁止接收器输出。
- DE：驱动器工作使能。引脚为 1，允许驱动器工作；引脚为 0，禁止驱动器工作。
- D：驱动器输入。
- GND：接地管脚。
- A：接收器非反向输入端和驱动器非反向输出端。
- B：接收器反向输入端和驱动器反向输出端。

- VCC：电源脚。电压范围 4.75V～5.25V。

经过对 SN75176 芯片的介绍，读者应该对 SN75176 有了基本的认识，下面介绍其他电路的具体设计。

## 17.3.2 232/485 转接卡

由于主控机是 PC，它只有标准的 RS-232 接口，因此需要一个 232/485 转接设备才能接入 485 总线网络。如图 17-3 所示为 232/485 转接卡的设计原理图。

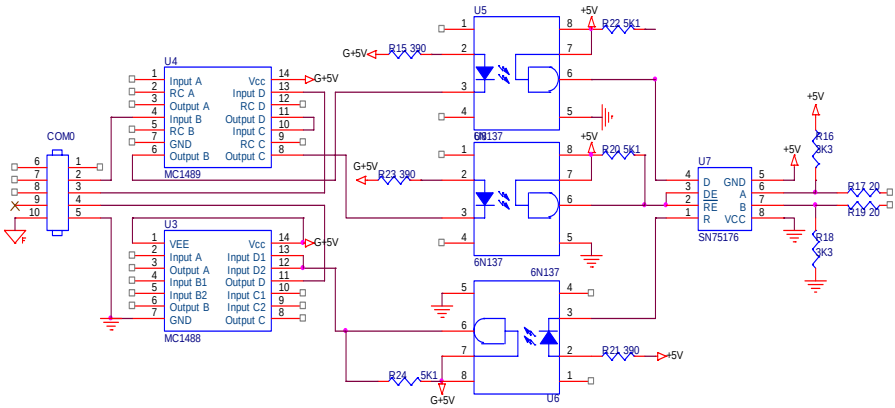



图 17-3 232/485 转接卡原理图

图 17-3 中，MC1488 和 MC1489 是摩托罗拉公司的芯片，它们的作用是实现 TTL 电平和 RS-232 电平的转化，MC1488 是驱动器，MC1489 是接收器。

6N137 是 Toshiba 公司的光电隔离器件，针对要使用的串行口信号使用了 3 块 6N137。

U？是 DC-DC 功能模块，它的作用是将电源隔离，降低直流电源的干扰。

U7 是 485 驱动收发芯片 SN75176，它是实现 232/485 转接的关键，它的 DE 和/RE 引脚直接连接，因为它是半双工芯片，不是驱动有效就是接收有效，二者不能同时有效。

## 17.3.3 带 485 接口的单片机系统

单片机选用 Freescale 公司的 MC9S08JM60。单片机系统的功能主要包括两部分：数据采集和与 485 总线的接口，这两部分可以独立设计，本章主要介绍与 485 的接口设计。

单片机系统的 485 接口仍然采用 SN75176 芯片。单片机 MC9S08JM60 的自带串口和 SN75176 相连，如图 17-4 所示。

如图 17-4 所示，单片机的串口引脚 RXD 和 TXD 分别连接 SN75176 的 R 和 DI 引脚，以进行串行数据交换；控制引脚 PTE2 接 SN75176 的

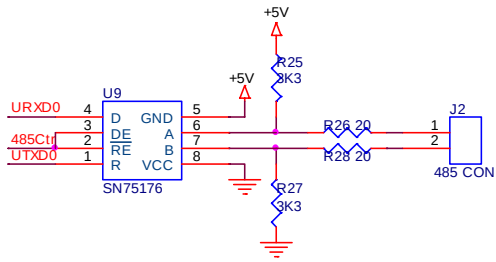



图 17-4 单片机系统的 485 接口电路图

DE 和/RE 引脚，以控制驱动器和接收器使能。注意这 3 个引脚均应接上上拉电阻。

SN75176 的 A 和 B 引脚为 485 总线网络的差分信号输入/输出端，二者之间应串接一个 120Ω 的电阻。

单片机 MC9S08JM60 工作在 12MHz 时钟下。图 17-5 中的其他部分电路为单片机的常见附属电路（例如仿真调试电路、复位电路）及和数据采集部分的电路连接，这里就不作说明了。

SW1 为一个 DIP 开关，和单片机 MC9S08JM60 的 PTB0~PTB5 引脚相连，用于设置本机的设备号。由于 SN75176 实现的 485 总线上最多带有 32 个负载，所以 6 位引脚足够使用。

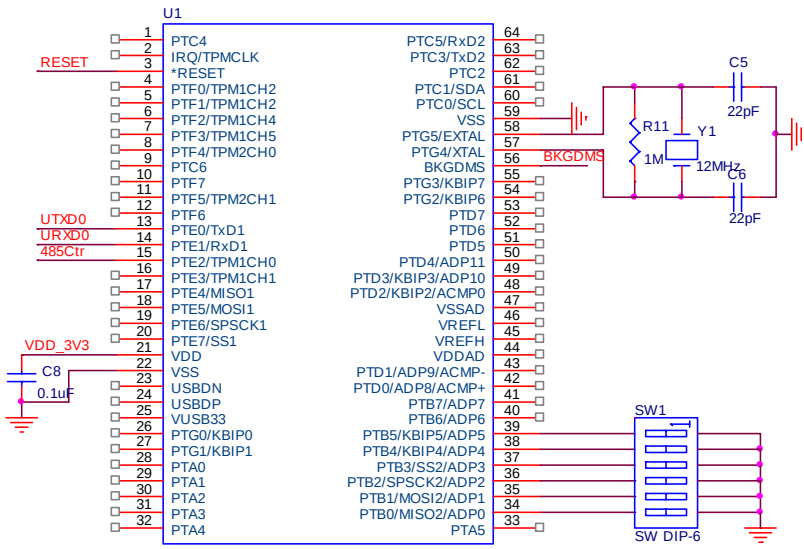



图 17-5　单片机系统电路图

## 17.4　软件设计与代码

### 17.4.1　软件设计流程

整个系统软件分为主控机（PC）端和单片机端两部分。

除了通信接口部分的软件以外，主控机端软件还包括用户界面、数据处理、后台数据库等。单片机端软件包括数据采集和 485 通信程序，这两部分可以完全独立，数据采集部分可设计成一个函数，在主程序中调用即可。

主控机端通信接口部分软件的流程如图 17-6 所示。

由图 17-6 可以看出，整个程序的流程和协议设计密切相关。

对应从机而言，它的工作与主机密切相关，它根据主机的指令执行相应的操作。从机何时去收集设备的状态信息也取决于主机，当从机收到主机发送读设备状态信息指令

“GETDATA”时，才开始收集信息并发送“SENDDATA”上报。

单片机系统的通信软件流程如图 17-7 所示。

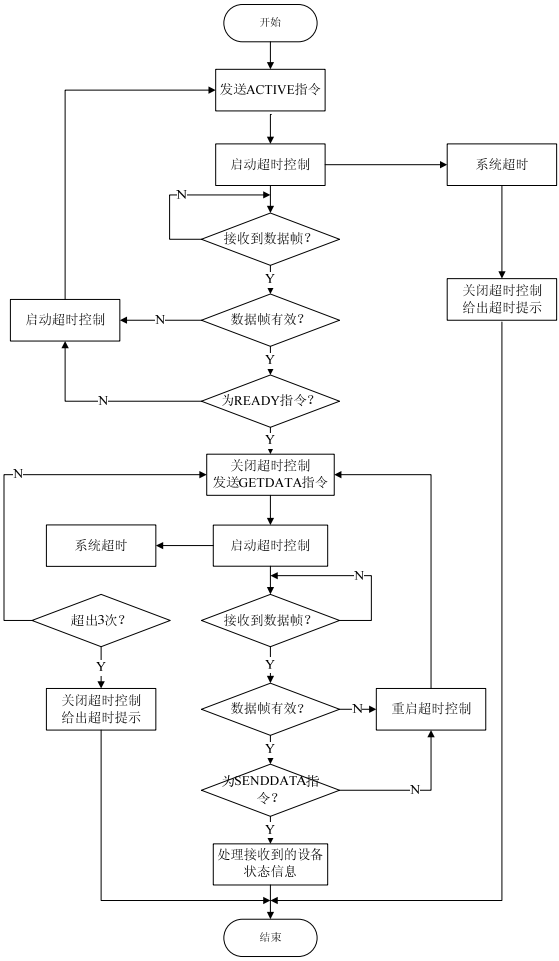



图 17-6　主控机端 485 通信接口部分软件流程

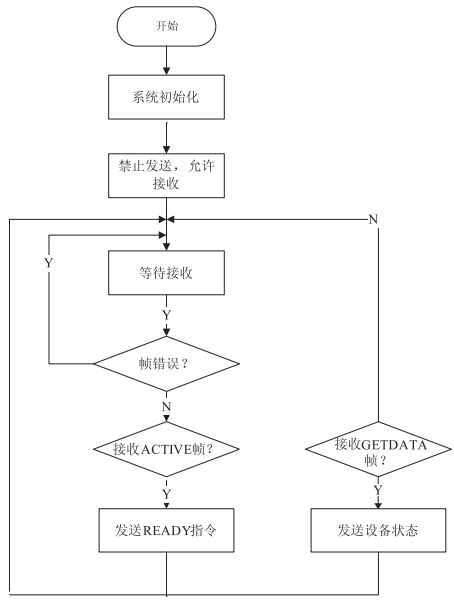



图 17-7　单片机端 485 总线通信软件流程

## 17.4.2　串口通信子程序

串口通信在此采用查询方式。下面介绍具体的程序。

```
//串口接收函数
uchar Uart_RecvChar(Uart_TComData *Chr)
{
  ucharbyte Result = ERR_OK;                    //错误标志
  ucharbyte StatReg = SCI1S1;                   //读状态寄存器

  if(StatReg &(SCI1S1_OR_MASK|SCI1S1_NF_MASK|SCI1S1_FE_MASK|
```

```
SCI1S1_PF_MASK))
{ //错误标志置位?
    Result = ERR_COMMON;
  }
else if (!(StatReg & SCI1S1_RDRF_MASK)) //接收器空，没有错误发生?
{
    return ERR_RXEMPTY;
  }
else
{
  }
  *Chr = SCI1D;                        //接收字符
  return Result;                       //返回错误码
}

//串口发送函数
ucharbyte Uart_SendChar(Uart_TComData Chr)
{
  if (!SCI1S1_TDRE)                   //发送器空?
{
    return ERR_TXFULL;                 //发送器满
  }
  SCI1D = (byte)Chr;                   //发送字符
  return ERR_OK;
}
```

### 17.4.3 485 通信收发子程序

485 通信的收发控制通过管脚 PTE2 控制。下面介绍具体的程序。

```
//接收数据帧函数，实际上接收的是主机的指令
uchar Recv_Data(uchar *type)
{
  uchar tmp,rCount,i;
  uchar r_buf[RECFRMMAXLEN];  //保存接收到的帧
  uchar Flag_ReceOver;        //一帧接收结束标志
  uchar Flag_StartRec;        //一帧开始接收标志
  uchar CheckSum;             //校验和
  uchar DataLen;              //数据字节长度变量

  Uart_Ctrl_ClrVal();         //禁止发送，允许接收

  //接收一帧数据
  rCount = 0;
  Flag_StartRec = 0;
```

```
Flag_ReceOver = 0;

while(!Flag_ReceOver)
{
  Uart_RecvChar(&tmp);

  if((!Flag_StartRec) && (tmp == 0x24)) //判断是否接收到字符“$”
  {
     Flag_StartRec = 1;
  }

  if(Flag_StartRec)
  {
    r_buf[rCount] = tmp;
    rCount++;

    if(tmp == 0x2A)                  //判断是否接收到字符“*”
    {
       Flag_ReceOver = 1;
    }
  }

  if(rCount == RECFRMMAXLEN)           //帧超长错误，返回 0
  {
     return 0;
  }
}

CheckSum = 0;                        //计算校验和字节
DataLen = r_buf[3];

for(i=0; i++; i<3+DataLen)
{
  CheckSum +=r_buf[i+1];
}

if(rCount < 6)                      //帧过短错误，返回 0，最短的指令帧为 6 个字节
  return 0;

if(r_buf[1] = DevNo)                //地址不符合错误，返回 0
  return 0;

if(r_buf[rCount -2] != CheckSum)    //校验错误，返回 0
  return 0;
```

```
  *type = r_buf[2];                //获取指令类型

  return 1;                        //成功，返回 1
}

//发送数据函数，包括发送指令和发送设备状态信息
void Send_Data(uchar type, uchar len, uchar *buf)
{
  uchar i,tmp;
  uchar CheckSum = 0;

  Uart_Ctrl_SetVal();              //允许发送，禁止接收

  tmp = 0x24;
  Uart_SendChar(tmp);              //发送帧起始字节

  Uart_SendChar(DevNo);            //发送地址字节，也即设备号
  CheckSum +=DevNo;

  Uart_SendChar(type);             //发送类型字节
  CheckSum +=type;

  Uart_SendChar(len);              //发送数据长度字节
  CheckSum +=len;

  for(i=0; i<len; i++)             //发送数据
  {
    Uart_SendChar(*buf);
    CheckSum +=*buf;
    *buf++;
  }

  Uart_SendChar(CheckSum);

  tmp = 0x2A;
  Uart_SendChar(tmp);              //发送帧结束字节
}
```

### 17.4.4 主程序

系统经过一系列初始化后，即进入主程序工作。主程序调用 485 通信子程序，并根据不同的命令执行相应的操作。下面给出现场监测系统的主程序。

```
void main(void)
```

```
{
  PE_low_level_init();
  DevNo = Device_Addr_GetVal();

  for(;;)
  {
    if(Recv_Data(&type) == 0) //接收帧错误或者地址不符合，丢弃
    {
      continue;
    }

    switch(type):
    {
      case ACTIVE:          //主机询问从机是否在线
        Send_Data(READY, 0, StatusBuf);//发送 READY 命令
        break;

      case GETDATA:        //主机读设备请求
        Clr_StatusBuf();    //设备状态信息缓冲区初始化
        Get_Stat();        //数据采集函数
        Send_Data(SENDDATA, strlen(StatusBuf), StatusBuf);
                  //发送设备状态信息
        break;

      default:
        break;

    }
  }
} /*** End
```

## 17.5 实例总结

本章详细介绍了基于 MC9S08JM60 单片机的 485 总线现场监测系统的软、硬件设计，主要侧重于 RS-485 总线网络的构建。读者在学习本例的设计实现过程中，有如下几个方面值得注意：

本例是采用 SN75176 芯片来实现 485 接口的，这决定了所实现的 485 总线网络是半双工通信网络；若是必须实现全双工，可以选择其他 485 驱动芯片。

485 总线未定义应用层协议，所以和通信相关的协议设计就成为软件设计的重点，通信协议的好坏直接决定了整个系统软件的性能。

# 第 18 章

# $I^2C$ 总线通信模块：串行存储器接口设计实例

在单片机应用中，可以利用串行存储器存储一些配置信息或者运行参数。本章将介绍基于MC9S08JM60单片机的串行存储器芯片24LC02的接口设计。首先介绍硬件电路设计，然后再介绍软件设计。

## 18.1 实例内容说明

MC9S08JM60 单片机内置 $I^2C$ 模块，利用单片机自带的 $I^2C$ 模块不仅很容易与串行存储器进行接口，而且接口非常简单。如图 18-1 所示为系统的原理框图。

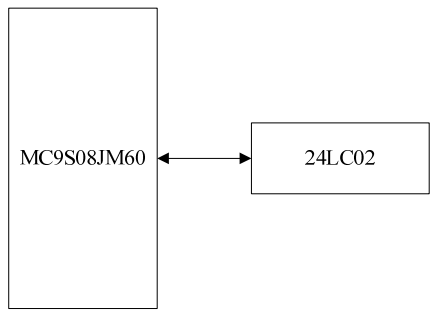



图 18-1 系统原理框图

由图 18-1 可以看出，系统的硬件电路非常简单。下面介绍具体的设计思路。

## 18.2 设计思路分析

串行存储器接口设计的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60，MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、$I^2C$ 控制器、定时器等多种外设，方便进行系统设计。

按照图 18-1 所确定的系统原理框图，关键在于 $I^2C$ 协议的实现，因此首先需要了解 $I^2C$ 通信协议的相关知识。

I²C 是由 Philips 公司开发的一种用于内部芯片控制的简单双向两线串行总线。I²C 总线支持任何一种芯片制造工艺，并且 Philips 和其他厂商提供了种类非常丰富的 I²C 兼容芯片。作为一个专利的控制总线，I²C 已经成为世界性的工业标准。

I²C 总线采用两线：串行数据线（SDA）和串行时钟线（SCL）在连接到总线的器件间传递信息。每个器件都有一个唯一的识别地址，而且都可以作为一个发送器或接收器（由器件的功能决定）。除了发送器和接收器外，器件在执行数据传输时也可以被看做主机或从机。作为主机时它是初始化总线的数据传输并产生允许传输的时钟信号的器件，此时任何被寻址的器件都被认为是从机。I²C 总线是一个多主机的总线，这就是说可以连接多于一个能控制总线的器件到总线。SDA 和 SCL 都是双向线路，都通过一个电流源或上拉电阻连接到正的电源电压，当总线空闲时，这两条线路都是高电平，连接到总线的器件输出级必须是漏极开路或集电极开路才能执行线“与”的功能。下面讲述 I²C 的特征。

- 只需要两线的总线线路：一条串行数据线（SDA）和一条串行时钟线（SCL）。
- 每个连接到总线的器件都可以通过唯一的地址和一直存在的简单主/从机关系软件设定地址，主机可以作为主机发送器或主机接收器。
- 它是一个真正的多主机总线，如果两个或更多主机同时初始化数据传输，可以通过冲突检测和仲裁来防止数据被破坏。
- 串行的 8 位双向数据传输位速率在标准模式下可达 100kbps，快速模式下可达 400kbps，高速模式下可达 3.4Mbps。
- 片上的滤波器可以滤去总线数据线上的毛刺波，保证数据完整。
- 连接到相同总线的 IC 数量只受到总线的最大电容 400pF 的限制。

由于连接到 I²C 总线的器件有不同种类的工艺（CMOS、NMOS、双极性），逻辑“0”（低电平）和“1”（高电平）的电平不是固定的，它由 VCC 的相关电平决定，每传输一个数据位就产生一个时钟脉冲。SDA 线上的数据必须在时钟的高电平周期保持稳定。数据线的高电平或低电平状态只有在 SCL 线的时钟信号是低电平时才能改变。

I²C 的时序必须包括启动信号、数据传输、确认和停止信号。下面对这几个部分进行简要介绍。

**1. 启动信号和停止信号**

在 I²C 总线中唯一出现的是被定义为启动和停止的信号。其中一种情况是在 SCL 线保持高电平期间，SDA 线上的电平负跳变被定义为 IIC 总线的启动信号。启动信号是一种电平跳变时序信号，而不是一种电平信号。启动信号是由主机主动建立的，在建立该信号之前，I²C 总线必须处于空闲状态，即 SDA 线和 SCL 线同时处于高电平。该信号表示开始新的数据传输，并使所有被控制器退出空闲状态。

在 SCL 线保持高电平期间，SDA 线被释放，使 SDA 线的电平正跳变，这被定义为 I²C 总线的停止信号。停止信号也是一种电平跳变时序信号，而不是一种电平信号。停止信号也是由主机主动建立的，在建立该信号后，I²C 总线返回空闲状态。在连续读时，如接收

到一个“停止信号”，则所有读操作将终止，芯片将进入等待模式。总线在启动信号后被认为处于忙的状态，在停止信号的某段时间后总线被认为再次处于空闲状态。

**2. 数据传输**

成功实现从机寻址之前，就可以按照主叫主机发送的 R/W 位指定的方向逐字节地进行数据传输。地址周期后的所有传输都被称为数据传输，即使它们包含从机的子地址报文。每个数据字节的长度均为 8 位。只有当 SCL 线为低电平时 SDA 线的状态才可以变化，如果 SCL 线处于高电平，那么 SDA 线的状态必须保持稳定。SCL 线的一个时钟脉冲传输一个数据位，最高位被首先传输。每个数据字节后面都有一个第 9 位（应答位），该位由从机发出信号，通过将 SDA 线的第 9 个时钟位拉低实现。总之，一个完整的数据传输需要 9 个时钟脉冲。

**3. 确认**

数据传输必须带确认信号，相关的确认时钟脉冲由主机产生，在确认的时钟脉冲期间发送器释放 SDA 线（高）。在确认的时钟脉冲期间，接收器必须将 SDA 线拉低，使它在这个时钟脉冲的高电平期间保持稳定的低电平，当然必须考虑建立和保持时间。通常被寻址的接收器在接收到的每个字节后必须产生一个确认。当从机不能确认从机地址时（如它正在执行一些实时函数不能接收或发送），从机必须使数据线保持高电平，然后主机产生一个停止条件终止传输或者产生重复起始条件开始新的传输。如果从机接收器确认了从机地址，但是在传输了一段时间后不能接收更多数据字节，则主机必须再一次终止传输。这个情况用从机在第一个字节后没有产生确认来表示，从机使数据保持高电平，主机产生一个停止或重复起始条件。如果传输中有主机接收器，主机必须在从机发送完最后一个字节时不产生确认信号，向从机发送器通知数据结束，从机发送器必须释放数据线，允许主机产生一个停止或重复起始条件。

## 18.3 硬件电路设计

本系统的硬件电路相对简单，主要有电源电路、复位电路、仿真调试电路、串行存储电路和单片机电路。在介绍具体的硬件电路前，先介绍一下 AT24C02 芯片。

### 18.3.1 AT24C02 芯片的介绍

AT24C02 是 Atmel 公司生产的 IIC 接口的 EEPROM，容量为 256×8 位。该芯片具有以下特性：

- 单电源供电，电压范围为 2.5V～5.5V。
- 低功耗。工作时，典型的电流值为 2mA，芯片空闲时，典型的电流值最大为 8uA。
- $I^2C$ 总线接口。
- 具有写保护功能，这样可以作为硬件数据保护。
- 可写多达 100 万次，数据可保存 100 年。

为了便于进行硬件电路的设计，下面给出该芯片的管脚图，如图 18-2 所示。

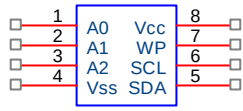



图 18-2　AT24C02 的管脚图

由图 18-2 可以看出，该芯片只有 8 个管脚。下面对具体的管脚进行介绍。

- A0～A2：芯片地址线。用来进行期间寻址。
- VSS：电源接地。
- SDA：串行数据输入/输出脚，此管脚通常使用一个电阻进行上拉。
- SCL：串行时钟输入脚，由于在 SCL 上升/下降沿处理信号，要特别注意 SCL 信号的上升/下降时间。此管脚通常用一个电阻进行上拉。
- WP：写保护管脚。该管脚接高电平，则处于写保护状态；该管脚接低电平，则可以进行读/写。
- VCC：电源管脚。

经过对串行存储器芯片 AT24C02 芯片的介绍，读者应该对串行存储器电路有了基本的认识，下面介绍具体的串行存储器电路。

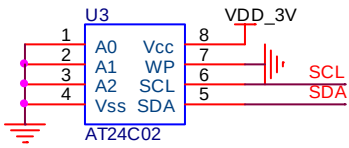



图 18-3　串行存储器的电路原理图

## 18.3.2　串行存储器电路

经过前面对 AT24C02 的介绍，该部分电路的设计应该不难。AT24C02 主要是通过 I²C 实现与单片机的连接，如图 18-3 所示为具体的电路原理图。

由图 18-3 可以看出该硬件电路设计简单。AT24C02 的第 7 管脚（写保护管脚）接地，使该芯片始终处于可以进行读/写的状态。在实际设计的时候，也可以将 WP 管脚与单片机的 1 个普通 I/O 端口进行连接，通过单片机来控制 AT24C02 的写保护状态，即单片机该管脚输出高电平，AT24C02 就处于写保护状态；单片机该管脚输出低电平，AT24C02 可以正常读写。在本电路中，为简化设计，直接将 WP 管脚接地。AT24C02 的 A0、A1 和 A2 都接地，表示该器件的地址为 00。由于 I²C 是总线工作方式，该总线上可以挂接很多器件，所以总线上的每个器件都应具有唯一的一个地址，这样才能实现寻址操作。AT24C02 的 SCL 和 SDA 管脚分别与单片机（MC9S08JM60）SCL 和 SDA 管脚进行连接，连接的方式是 I²C 总线方式。在设计时，需要将 SCL 和 SDA 分别通过一个 10kΩ 的电阻进行上拉，以满足 I²C 工作的条件。此外，为了减少电源的干扰，还需要在 AT24C02 芯片的电源输入管脚加一个 0.1uF 的电容来实现滤波，以减少输入端受到的干扰。

## 18.3.3　单片机电路

单片机选用 Freescale 公司的 MC9S08JM60，单片机系统的功能就是和 AT24C02 串行存储器实现 I²C 接口，电路如图 18-4 所示。

单片机 MC9S08JM60 工作在 12MHz 时钟下。图 18-4 中的其他部分电路为单片机的常见附属电路（例如仿真调试电路、复位电路），这里就不作说明了。

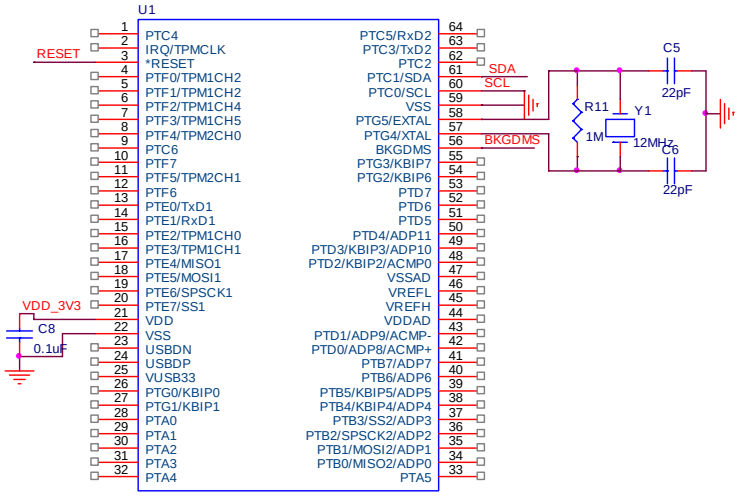



图 18-4　单片机系统电路图

# 18.4　软件设计与代码

## 18.4.1　软件设计流程

串行存储接口设计的系统软件就是 AT24C02 的读/写程序，对 AT24C02 的读/写是通过 $I^2C$ 实现的。其读/写程序的流程图如图 18-5 所示。下面分别介绍具体的程序。

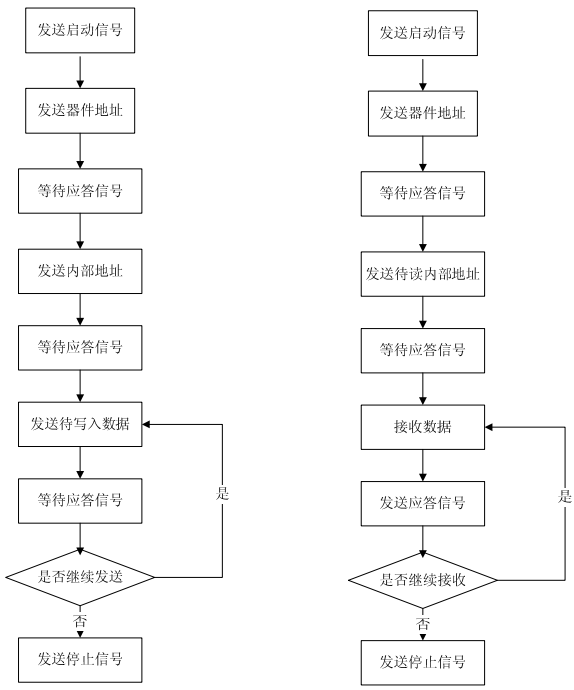



图 18-5　AT24C02 读/写程序流程图

## 18.4.2　I²C 模块初始化程序

I²C 模块初始化程序主要通过设置相应的寄存器实现对 I²C 模块的设置。在完成了初始化操作后，就可以对 AT24C02 进行读/写操作了。下面介绍具体的初始化程序。

```
void IIC_init(void)
{
    uchar temp;
    IICA=0X00;
    IICF=0x5b;      //设置波特率 RATE=100k bps
    IICC1_IICEN=1;  //使能 IIC 模块
    IICC1_IICIE=0;  //除能 IIC 中断
    IICC1_MST=0;    //设置从模式
    IICC1_TXAK=1;   //设置产生应答
    IICS_BUSY=1;    //总线忙标志置 1
    IICS_SRW=0;     //设置为发送模式
    IICS_ARBL=0;    //仲裁标志置 0
    IICS_IICIF=0;   //发送标志置 0
    temp=IICD;
}
```

## 18.4.3　AT24C02 的读/写子程序

AT24C02 的操作主要包括读操作和写操作。下面对这两种操作进行具体介绍。

**1. 读操作**

根据 AT24C02 的读时序可知：单片机首先发送一个控制字节，然后发送地址数据，最后读出该地址的数据。下面给出具体程序。

```
void IIC_read_block(char addr,char len)
{
   IICC1_IICEN=0;  //IIC 模块除能
   Address=addr;    //读取数据的首地址
   length=len;      //读取数据的长度
   i=0;            //清零数据访问的循环变量
   IIC_Res_flg=0;   //清零传输完成的标志
   flag=10;        //选择不同功能的中断处理
   if(length>16)    //限定读取数据的长度
   {
    length=16;
   }
   IICC1_TXAK=0;   //设置产生应答信号
   IICC1_IICEN=1;  //使能 IIC 模块
   IICC1_MST=1;    //设置为主机模式
   IICC1_TX=1;     //设置为发送模式
```

```
  IICC1_IICIE=1;  //使能 IIC 中断
  IICD=IIC_SLAVE&0xfe; //首先发送从机地址
 }
```

**2. 写操作**

根据 AT24C02 的写时序可知：单片机首先发送控制字节，然后发送地址字节，最后输入写的内容。下面为具体的写操作程序。

```
void IIC_write_block(char addr,char len)
 {
  IICC1_IICEN=0;  //IIC 模块除能
  Address=addr;    //写入数据的首地址
  length=len;      //写入数据的长度
  i=0;            //清零数据访问的循环变量
  IIC_Res_flg=0;   //清零传输完成的标志
  flag=2;         //选择不同功能的中断处理
  if(length>16)    //限定写入数据的长度
  {
   length=16;
  }
  IICC1_TXAK=1;   //不产生应答信号
  IICC1_IICEN=1;  //使能 IIC 模块
  IICC1_MST=1;    //设置主机模式
  IICC1_TX=1;     //设置为发送模式
  IICC1_IICIE=1;  //使能 IIC 中断
  IICD=IIC_SLAVE&0xfe; //首先发送从机地址
}
```

**3. 中断服务程序**

下面为具体的中断服务程序。

```
ISR(I2C_isr) {
 uchar temp;
 IICS_IICIF=1;  //清零中断标志
 switch(flag){
 //读取功能
   case 2:
   {
     IIC2D=Address;
     flag++;return;
   } //首先发送读取的起始地址
   case 3:
   {
     if(length)
     {
```

```
      IICD=data[i++];
      length--;
      return;
     }//开始读取数据
     else
     {
       IICC1_MST=0;
       IIC_Res_flg=1;
       IICC1_IICIE=0;
       return;
     }
          //读取数据完毕，重新进入发送数据模式，置为传输完成标志，除能 IIC 模块
   }
//写入功能
  case 10:
  {
    IICD=Address;
    flag++;
    return;
  }                  //先发读取的存储首地址
  case 11:
  {
    IICC1_RSTA=1;
    IICD=IIC_SLAVE|1;
    flag++;
    return;
   }  //RSTA 重新开始信号，无须应答
      //发送从机地址，启动读取 R/W=1
  case 12:
  {
    IICC1_TX=0;
    IICC1_TXAK=0;
    temp=IICD;
    flag++;
    return;
  }//开始接收，第一个为应答信号
  case 13:
  {
    if(length-1)
    {
      data[i++]=IICD;
      length--;
      return;
    }     //接收数据
```

```
      else
      {
        data[i++]=IICD;
        IICC1_TXAK=1;
        flag++;
        return;
      }//最后一位数据不需应答信号
    }
    case 14:
    {
      data[i++]=IICD;
      IICC1_MST=0;
      flag=0;
      IIC_Res_flg=1;
      IICC1_IICIE=0;
      IICC1_IICEN=0;
      return;
     //写入数据完毕，重新进入发送数据模式，清零中断功能选择标志，置为传输完成标志
     //除能 IIC 模块
    }
  }
}
```

### 18.4.4 主程序

系统经过一系列初始化后，即进入主程序工作。主程序调用前面给出的读/写子程序，实现串行存储接口设计的程序。下面给出主程序。

```
void main(void)
{
   PE_low_level_init();
   IIC_init();                  //初始化 IIC 模块
  IIC_read_block(0x20,12);      //读取 IIC 存储芯片数据
  delay(10000);
  while(length,IIC_Res_flg==0); //等待数据读取完成
  for(m=0;m<16;m++)
     data[m]=200-5*m;           //写入待发送的数据
  IIC_write_block(0x20,16);     //写入数据到 IIC 存储芯片
  for(m=180;m>0;m--)
      delay(60000);
  while(length,IIC_Res_flg==0); //等待数据写入完成
  delay(60000);
  for(m=0;m<10;m++)             //清零数据数组
     data[m]=0x0;
  delay(10000);
```

```
    IIC_read_block(0x20,16);      //重新读取 IIC 存储芯片数据
    while(length,IIC_Res_flg==0);
    delay(60000);
    for(;;){}
  }
```

## 18.5　实例总结

本章详细介绍了基于 MC9S08JM60 单片机的串行存储器接口的软、硬件设计过程。由于 MC9S08JM60 单片机自带 I²C 模块，因此在软件上很容易实现。采用 I²C 总线接口，很容易实现与不同容量的串行存储器接口，而且不必改变硬件设计。对于大容量的串行存储器来说，一般只需要修改发送的地址数据就可以了。因此，本系统在软件和硬件上很容易实现升级。

# 第 19 章

# $I^2C$ 总线通信模块：$I^2C$ 主从通信系统设计实例

在单片机应用中有时需要进行多机通信，本章将介绍基于 MC9S08JM60 单片机 $I^2C$ 方式实现的主从多机通信系统，相对第 18 章的设计而言，设计思路与过程略微复杂一些。

## 19.1 实例内容说明

$I^2C$ 总线是由 Philips 公司开发的一种双向两线主机总线，它能方便地实现芯片间的数据传输与控制。通过 $I^2C$ 总线，可以方便地完成多机间的主从通信。基于 $I^2C$ 总线的多机通信电路结构简单、程序编写方便，易于实现系统软硬件的模块化和标准化。如图 19-1 所示为基于 $I^2C$ 总线实现的主从通信系统的原理框图。

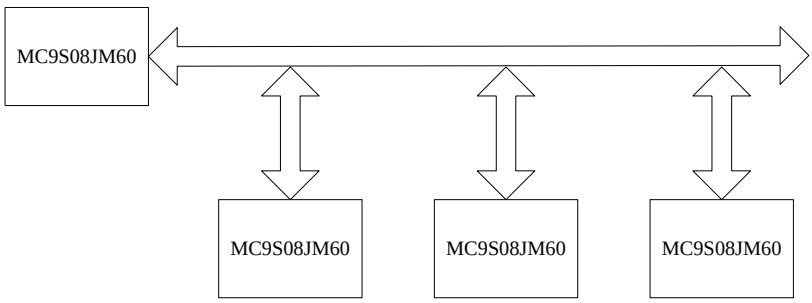



图 19-1　基于 $I^2C$ 总线实现的主从通信系统的原理框图

由图 19-1 可以看出，基于 I²C 总线实现的主从通信系统结构相对简单，扩展也较为容易。

## 19.2　设计思路分析

基于 I²C 总线实现的主从通信系统的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、I²C 控制器、定时器等多种外设，方便进行系统设计。

分析图 19-1 所确定的系统原理框图可知，关键在于 I²C 协议的实现。关于 I²C 通信协议的相关知识读者可参考第 18 章，下面介绍主从多机通信的相关知识。

由于 I²C 总线是双向总线，因此总线上的任何一个设备都是平等的，都可以向总线上发送数据。虽然总线上的设备可以随时向总线发送数据，但是总线上不允许有两个设备同时发送数据，如果有这种情况发生，就会引起总线竞争。I²C 总线通过总线仲裁来解决总线竞争问题。

如果有两个或者两个以上的设备向 I²C 总线同时发送数据时，就会引起 I²C 总线的竞争，此时就需要对 I²C 总线进行仲裁处理。由于 I²C 总线是通过上拉电阻进行上拉的，因此当总线空闲时，两根线都是高电平。此外，由于连接总线的器件输出极必须是集电极或漏极开路，因此 I²C 总线具有线“与”功能。通过 I²C 总线的线“与”功能，I²C 能很好地解决 I²C 总线的竞争问题。

I²C 总线上的仲裁分两部分：SCL 线的同步和 SDA 线的仲裁。SCL 同步是由于总线具有线“与”的逻辑功能，即只要总线上有一个设备发送低电平时，总线上就表现为低电平。当总线上的所有设备都发送高电平时，总线才能表现为高电平。正是由于这种线“与”功能，当总线上的多个设备同时发送时钟信号时，在总线上的时钟信号才能同步。这就是 SCL 的同步原理。

SDA 线的仲裁也是建立在总线具有线“与”逻辑功能的原理上。总线上的某个设备在发送 1 位数据后，比较总线上所呈现的数据与自己发送的是否一致。如果是一致的，则该设备继续发送数据；如果不一致，则该设备停止发送数据，并且退出竞争。

SDA 线的仲裁可以在 I²C 总线上的多个主设备企图控制总线时，保证通信正常进行并且数据不丢失。I²C 总线通过仲裁只允许一个主设备继续占据总线发送数据。下面简要介绍仲裁处理。

在 I²C 总线上进行数据传输时，SDA 为 I²C 总线上所呈现的数据信号，SCL 是 I²C 总线上的时钟信号。假定主设备 1、2 同时发送起始信号时，两个主设备都发送了高电平信号。这时总线上呈现的信号为高电平，两个主设备都检测到总线上的信号与自己发送的信号相同，继续发送数据。在第 2 个时钟周期，2 个主设备都发送低电平信号，在总线上呈现的信号为低电平，两个主设备都检测到总线上的信号与自己发送的信号仍然相同，则继续发送数据。在第 3 个时钟周期，主设备 1 发送高电平信号，而主设备 2 发送低电平信号。

根据 I²C 总线的线“与”功能，I²C 总线上的信号为低电平，这时主设备 1 检测到总线上的数据和自己发送的数据不一样，就停止数据的发送，将本设备设置成从机模式。这样主设备 2 就赢得了总线，并继续占有总线，并且数据没有丢失，即总线的数据与主设备 2 所发送的数据一样，而主设备 1 在设置为从设备后继续接收数据，同时也没有丢掉 SDA 线上的数据。

根据以上的介绍可知，在仲裁过程中，I²C 总线上的数据不会丢失。

## 19.3 硬件电路设计

本系统的硬件电路相对简单，主要有电源电路、复位电路、仿真调试电路和单片机电路。下面具体介绍单片机系统电路的设计。

单片机选用 Freescale 公司的 MC9S08JM60，由于该单片机自带 I²C 模块，因此在硬件设计时非常简单，只需将 PTC0 管脚和 PTC1 管脚设置成 I²C 模块功能管脚即可。PTC0 管脚和 PTC1 管脚就构成了 I²C 总线。由于 MC9S08JM60 单片机 I/O 口内置上拉电阻，可以通过软件来使能或禁止，因此在硬件设计时，PTC0 管脚和 PTC1 管脚无需上拉电阻。如图 19-2 所示为具体的单片机系统电路图。

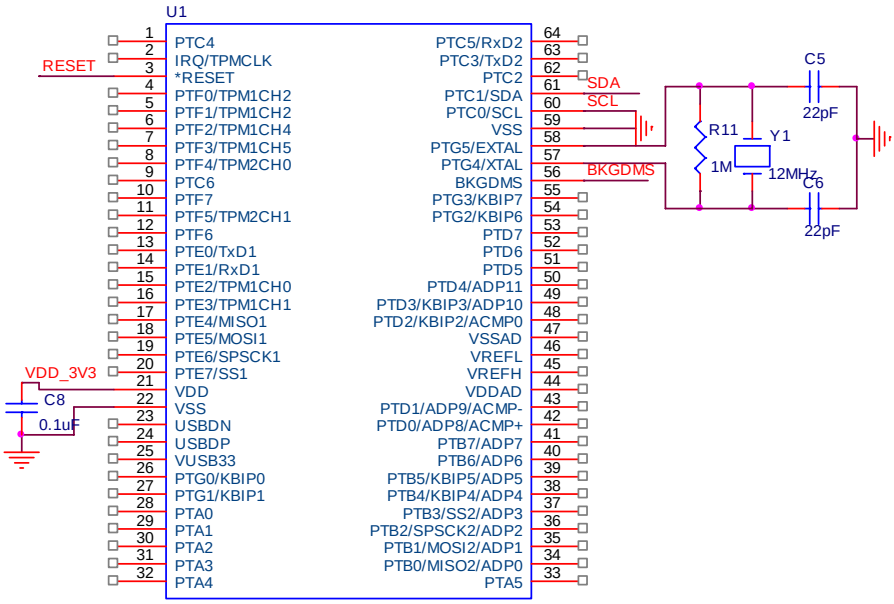



图 19-2　单片机系统电路图

由图 19-2 可以看出，该单片机电路非常简单。本系统中，由于是多机通信，因此 I²C 总线上有多个单片机，其他单片机的电路图与图 19-2 完全一致。考虑到电源的输入纹波对单片机的影响，在电源的管脚上增加一个 104 的滤波电容，以减少干扰。

# 19.4　软件设计与代码

## 19.4.1　软件设计流程

I²C 主从通信系统的软件设计主要包括两个部分：I²C 程序和测试程序。下面分别对每个部分进行具体的介绍。

## 19.4.2　I²C 程序

I²C 程序主要包括初始化程序、主机模式程序、从机模式程序和中断服务程序。下面给出每个部分的实现程序。I²C 初始化程序可以参考第 18 章的相关章节，这里不再介绍。

**1. 主机模式程序**

主机模式程序主要是实现主机模式下对总线的读/写操作，即数据的发送和接收。该部分程序分别包括发送初始化、接收初始化、发送数据和接收数据操作函数。

发送初始化和接收初始化程序主要设置 I²C 模块工作为主机模式，设置控制字节的内容，并设置相应的中断使能。下面为具体的程序。

```
void I²C_Master_Init(void)
{
  I²C_Master_SerFlag = 0x80;          //复位标志变量
  I²C_Master_SlaveAddr = 0x10;        //设置从机地址
  setReg8(IICF, 0x89);                //设置 SCL 时钟速率
  IICC1_IICEN = 1;                    //使能器件
  IICC1 = 0xC0;                       //设置控制寄存器
}

byte I²C_Master_SendBlock(void * Ptr,word Siz,word *Snt)
{
  if (!Siz) {                        //检查发送数据个数是否为 0
    *Snt = 0;
    return ERR_OK;
  }

  if((IICS_BUSY)||(InpLenM)||(I²C_Master_SerFlag&(CHAR_IN_TX|WAIT_RX_C
     HAR|IN_PROGRES)))
  {                                  //总线是否忙?
    return ERR_BUSOFF;
  }
  EnterCritical();                    //关中断
  I²C_Master_SerFlag |= IN_PROGRES;
  OutLenM = Siz;                     //设置数据字节值
  OutPtrM = (byte *)Ptr;              //保存待发送数据指针
```

```
  IICC1_TX = 1;                        //设置发送模式
  if(IICC1_MST) {                      //检查设备是否是主模式?
    IICC1_RSTA = 1;
  }
  else {
    IICC1_MST = 1;                     // 若不是，则生成启动信号
  }
  IICD = I2C_Master_SlaveAddr;         //发送从机地址
  ExitCritical();                      //开中断
  *Snt = Siz;
  return ERR_OK;
}

byte I2C_Master_RecvBlock(void* Ptr,word Siz,word *Rcv)
{
  if (!Siz) {                          //检查接收数据个数是否为 0?
    *Rcv = 0;
    return ERR_OK;
  }

  if((IICS_BUSY)||(InpLenM)||(I2C_Master_SerFlag&(CHAR_IN_TX|WAIT_RX_C
  HAR|IN_PROGRES)))                    //总线是否忙?
  {
    return ERR_BUSOFF;
  }
  EnterCritical();                     //关中断
  InpLenM = Siz;                       //设置接收数据字节数
  InpPtrM = (byte *)Ptr;               //保存待接收数据指针
  IICC1_TX = 1;                        //设置发送模式
  if(IICC1_MST) {                      //设备处于主模式?
    IICC1_RSTA = 1;
  }
  else {
    IICC1_MST = 1;
  }
  IICD = (byte)(I2C_Master_SlaveAddr+1); //发送从机地址
  ExitCritical();
  *Rcv = Siz;
  return ERR_OK;
}
```

**2. 从机模式程序**

从机模式程序主要是实现从机模式下对总线的读/写操作，即数据的发送和接收，该部

分程序分别包括发送初始化、接收初始化、发送数据和接收数据操作函数。下面为具体的程序。

```
void I²C_Slave_Init(void)
{
  I²C_Slave_SerFlag = 0;              //复位标志变量
  InpLenS = 0;                        //接收缓冲器清零
  InpPtrWS = InpPtrRS = 0;            //接收缓冲器索引清零
  OutLenS = 0;                        //发送缓冲器数据个数清零
  OutPtrWS = OutPtrRS = 0;            //发送缓冲器索引清零
  IICA = 0x04;                        //设置从机地址
  IICC2 = 0x00;
  setReg8(IICF, 0x89);                //设置 SCL 时钟速率
  IICC1_IICEN = 1;                    //使能设备
  IICC1 = 0xC0;                       //设置控制寄存器
}

byte I²C_Slave_SendBlock(void * Ptr,word Siz,word *Snt)
{
  word count;                         //发送数据字节数

  if (!Siz) {                         //检查待发送数据字节数是否为 0?
  *Snt = 0;
    return ERR_OK;
  }
  EnterCritical();
  *Snt = 0;                           //返回发送数据字节数
  for(count=0; count < Siz; count++)
  {
    if(OutLenS < 8)                   //发送器缓冲器未超出?
    {
      OutLenS++;
      OutBufferS[OutPtrWS++] = *((byte *)Ptr + count);//存储数据到缓冲器
      if (OutPtrWS >= 8)              //指针是否超出发送缓冲器?
      {
        OutPtrWS = 0;
      }
      (*Snt)++;
    }
    else {
      ExitCritical();
      return ERR_TXFULL;
    }
  }
```

```
  ExitCritical();
  return ERR_OK;
}

byte I²C_Slave_RecvBlock(void* Ptr,word Siz,word *Rcv)
{
  word count;                          //发送数据个数
  byte errrc = ERR_OK;
  if (!Siz) {                          //检查待接收数据个数是否为 0?
    *Rcv = 0;
    return ERR_OK;
  }
  EnterCritical();                      //关中断
  if ((I²C_Slave_SerFlag & (OVERRUN_ERR | FULL_RX)) != 0) //发生错误?
  {
    errrc = ERR_OVERRUN;
  }
  I²C_Slave_SerFlag &= ~(OVERRUN_ERR | CHAR_IN_RX | FULL_RX);
  *Rcv = 0;
  for(count=0; count < Siz; count++)
  {
    if(InpLenS) {                      //接收缓冲器是否溢出?
      InpLenS--;
     *((byte *)Ptr + count) = InpBufferS[InpPtrRS++]; //读数据
      if (InpPtrRS >= 8) {             //接收缓冲器索引值超出?
        InpPtrRS = 0;
      }
      (*Rcv)++;
    }
    else {
      ExitCritical();                   //开中断
      if(errrc==ERR_OVERRUN) {
        return ERR_OVERRUN;
      }
      else {
        return ERR_RXEMPTY;
      }
    }
  }
  ExitCritical();
  return errrc;
}
```

**3. 中断服务程序**

在主机模式或者从机模式下，数据的发送和接收都是通过中断服务程序实现的，下面

为具体的程序。

```
ISR(I²C_Master_Interrupt)
{
  byte Status;                          //状态寄存器变量

  Status = IICS;
  IICS_IICIF = 1;                       //清除中断标志位
  if(IICC1_MST) {                       //设备是否处于主模式?
    if(IICC1_TX) {                      //设备是否处于发送模式?
      if(Status & RXAK) {               //是否接收到 NACK?
        IICC1_MST = 0;                  //切换设备到从模式
        IICC1_TX = 0;                   //切换到接收模式
        OutLenM = 0;
        InpLenM = 0;
        I²C_Master_SerFlag &= ~(CHAR_IN_TX|WAIT_RX_CHAR|IN_PROGRES);
      }
      else {
        if(OutLenM) {                   //有字符发送?
          OutLenM--;
          IICD = *(OutPtrM)++;          //发送字符
        }
        else {
          if(InpLenM) {                 //有字符需接收?
            if(InpLenM == 1) {          //只需接收一个字符?
              IICC1_TXAK = 1;           //发送 ACK 禁止
            }
            else {
              IICC1_TXAK = 0;           //发送 ACK 使能
            }
            IICC1_TX = 0;               //切换到接收模式
            (void)IICD;
          }
          else {
            I²C_Master_SerFlag &= ~IN_PROGRES; //清除"busy"标志位
            IICC1_MST = 0;              //切换设备到从模式
            IICC1_TX = 0;               //切换到接收模式
            I²C_Master_OnTransmitData(); //调用发送数据事件
          }
        }
      }
    }
    else {
      InpLenM--;
```

```
      if(InpLenM) {                    //有字符需接收?
        if(InpLenM == 1) {
          IICC1_TXAK = 1;              //发送 ACK 禁止
        }
      }
      else {
        IICC1_MST = 0;
        IICC1_TXAK = 0;                //发送 ACK 使能
      }
      *(InpPtrM)++ = IICD;             //接收字符
      if(!InpLenM) {
        I2C_Master_OnReceiveData(); //调用接收数据事件
      }
    }
  }
  else {
    if(Status & IBAL) {                //仲裁丢失?
      OutLenM = 0;
      InpLenM = 0;
      I2C_Master_SerFlag &= ~(CHAR_IN_TX|WAIT_RX_CHAR|IN_PROGRES);
      IICC1_TX = 0;                    //切换到接收模式
    }
  }
}
```

### 19.4.3 主程序

系统经过一系列初始化后，即进入主程序工作。主程序调用前面介绍的程序，简单演示了主机模式和从机模式下数据的发送和接收。在具体应用时，读者可以根据自己系统的需要来设计相应的程序。下面给出 IIC 主从通信系统主机的主程序和从机的主程序。

**1. 主机主程序**

```
void main(void)
{
  PE_low_level_init();
  I2C_Master_SelectSlave(0x0a);   //设置从机地址

 PTAPE_PTAPE2 =1;    //Enable Pull ups on SW pins
 PTAPE_PTAPE3 =1;
  /* Write your code here */
  for(;;) {
     //选择通道
     if (PTAD_PTAD2 == 0)
     {
```

```
        while (PTAD_PTAD2 ==0);  //等待按键释放

         IIC_Data [0] = 0x01;    //设置阈值
         IIC_Data [1] = 0xff;
         IIC_Data [2] = 0x00;     //选择通道

        while (IICC_MST == 1);    //等待 IIC 停止
        //传输命令和阈值
        I²C_Master_SendBlock(&IIC_Data, 0x03, &bytes_to_trans);
      }

      if (PTAD_PTAD3 == 0)
      {
        while (PTAD_PTAD3 ==0);
         IIC_Data [0] = 0x01;
         IIC_Data [1] = 0xff;
         IIC_Data [2] = 0x01;
        while (IICC_MST == 1);
        I²C_Master_SendBlock(&IIC_Data, 0x03, &bytes_to_trans);
      }
      while (IICC_MST == 1);
      I²C_Master_RecvBlock(&IIC_Rec_Data, 0x03, &rec_count);

      if (IICS_BUSY == 0)
      {
        if (IIC_Rec_Data[2] == 0x01)
        {
            //Turn on Alarm Indicator
            Alarm = 0x01;
        } else
        {
           Alarm =0x00;
        }
      }
   }
}
```

**2. 从机主程序**

```
void main(void)
{
  PE_low_level_init();
  PTAPE_PTAPE4 = 1;
  PTAPE_PTAPE5 = 1;
```

```
  IIC_TX_Data [0] = 0x55;
  IIC_TX_Data [1] = 0x66;
  IIC_TX_Data [2] = 0x77;
  /* Write your code here */
  for(;;) {
    //检查哪个通道选中
     if (IIC_Rec_Data[2] == 0x01) {

      ADCSC1_ADCH = 1;
  //  Enter_Wait;
      IIC_TX_Data [0] = ADC_val_H;
      IIC_TX_Data [1] = ADC_val_L;
      IIC_TX_Data [2] = 0x50;

    if (ADC_val_H>>IIC_Rec_Data[0])
    {
              IIC_TX_Data [2] = 0x01;
    }
    else if (ADC_val_H==IIC_Rec_Data [0])
    {

       if (ADC_val_L>>IIC_Rec_Data [1] )
       {
           IIC_TX_Data [2] = 0x01;
        }
        else
        {
            IIC_TX_Data [2] = 0x50;
        }
    }
   }
   else  if (IIC_Rec_Data[2] == 0x00)
    {
       ADCSC1_ADCH = 0;
//       Enter_Wait;
       IIC_TX_Data [0] = ADC_val_H;
       IIC_TX_Data [1] = ADC_val_L;
       IIC_TX_Data [2] = 0x50;

        if (ADC_val_H>>IIC_Rec_Data[0])
        {
            IIC_TX_Data [2] = 0x01;

        }
```

```
        else if (ADC_val_H==IIC_Rec_Data [0])
        {

            if (ADC_val_L>>IIC_Rec_Data [1] )
            {
               IIC_TX_Data [2] = 0x01;
            }
            else
            {
               IIC_TX_Data [2] = 0x50;
            }
      }
    }
  }
}
```

## 19.5　实例总结

本章详细介绍了基于 MC9S08JM60 单片机的 I²C 总线实现的主从多机通信。利用这种方式实现的多机通信系统在硬件上并不复杂，而且程序易于移植，非常利于实现系统软/硬件的模块化和标准化。学习完本章实例后，读者可以在本章介绍的基础上，根据自己系统的需要实现更加复杂的系统。

# 第 20 章

# SPI 总线通信模块：电量监测仪设计实例

随着单片机技术、通信技术和计算机网络技术在工业生产过程中的不断推广和应用，串行总线微处理器和外界接口芯片构成的串行单片机系统也越来越多。本章将介绍基于 MC9S08JM60 单片机 SPI 模块实现的简易电量监测仪。

## 20.1 实例内容说明

SPI 总线是 Motorola 公司提出的一种同步串行外围总线系统，它可以使微处理器与各种外围设备以同步串行方式进行信息交换。采用 SPI 总线，可以很方便地与很多厂家生产的标准外围器件直接接口。这类器件很多，既可以是简单的 TTL 移位寄存器，也可以是复杂的 LCD 显示驱动器或 A/D 转换子系统。如图 20-1 所示为基于 SPI 模块实现的简易电量监测仪的原理框图。

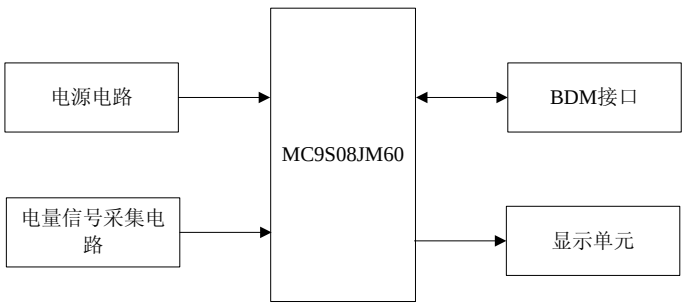



图 20-1　基于 SPI 模块实现的电量监测仪的原理框图

由图 20-1 可以看出，基于 SPI 模块实现的电量监测仪系统可分为 3 个部分：电量信号采集部分、显示单元和单片机系统。其中电量信号采集部分主要负责电量信息的采集，如线路的电压和电流信号；显示单元负责显示最终计算出的电压、电流值；单片机负责整个监测仪的工作，读取电量采集部分的结果，经处理后，将结果显示在显示单元上。

## 20.2　设计思路分析

基于 SPI 模块实现的电量监测仪的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、$I^2C$ 控制器、定时器等多种外设，方便进行系统设计。

按照图 20-1 所确定的系统原理框图，关键在于 SPI 协议的实现和电能计量芯片的选择，因此首先需要了解 SPI 通信协议的相关知识。

### 20.2.1　SPI 协议

SPI 系统的工作原理好像一个分布式 16 位移位寄存器，一半在微控制器里（即 SPI），另外一半在外设里。当控制器准备好发送数据时，这个分布式 16 位寄存器循环位移 8 个位，这样就有效地在微控制器与外设之间交换了数据。在某些情况下，这种循环位移是不完全的，因为数据可能只是从微控制器到外设或从外设到微控制器。

一个典型的 SPI 系统包括一个主控制器和一个或多个从属的外设。也有一些系统由两个或多个主微控制器，但这种配置很少见。很多制造商都生产同步串行外设，为了兼容大多数外设，SPI 系统可以选择移位率、主从模式及时钟的极性、相位等。MC9S08JM60 微控制器总线不可扩展，这就使很多用户感到不便。其实用户可以通过 SPI 系统进行串行扩展，带上 RAM、ROM、A/D、LCD 控制器等很多外设，而且比总线扩展更简单。

SPI 特性描述如下：

- 全双工，三线同步传送。
- 主机或从机工作方式。
- 使用相互独立的发送和接收数据寄存器实现双缓存操作。
- 4 种可编程主机移位频率（最大可达总线频率的二分之一）。
- 从机的最大移位频率等于总线频率。
- 可编程时钟极性和相位。
- 发送结束中断标志 SPTE，接收结束中断标志 SPRF。
- 工作方式设置错误中断标志 MODF。
- 溢出错误中断标志 OVRF。
- 可编程线或模式。
- 与 $I^2C$ 兼容，一个 SPI 子系统可以在软件控制下构成复杂或简单配置。

- 一个主控制器和几个从微控制器相互连接。
- 几个微控制器相互连接构成一个多主机系统。
- 一个主微控制器和一个或多个从属外设相互连接。

SPI 系统最常见的应用是一个微控制器做主机。在这种配置下，主机发起并控制数据的传送和流向，只有在主机发出通知后，从属设备才能从主机读取数据或向主机发送数据。本章的所有讨论都建立在这种配置之下。

**1. SPI的数据流动**

在 SPI 的应用中，不需要为每次传送的数据特别指出数据流动的方向，因为主机用两根独立的引脚来同时接收和发送数据。分布式移位寄存器操作如图 20-2 所示，在一次 SPI 传送发生时，一个 8 位字节通过“主出从入”MOSI 引脚移位输出，同时另一个 8 位字节从“主入从出”MISO 引脚输入。这个传送过程可以理解为主机的 8 位移位寄存器和从机电路联合组成了一个 16 位循环移位寄存器。在每次传送发生时，这个 16 位分布式移位寄存器循环移动 8 位，这样就完成了主机和外设之间的数据交换。

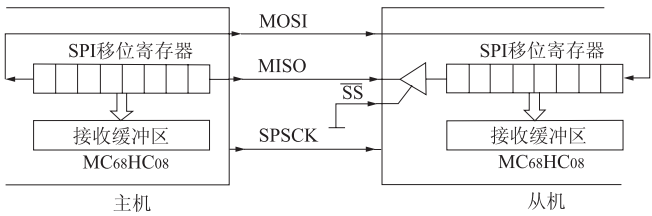



图 20-2 分布式移位寄存器操作图

**2. SPI功能**

SPI 系统用 D 端口的 4 根 I/O 引脚来实现数据传送：串行时钟 SPSCK/、主入从出 MISO/、主出从入 MOSI/ 和低有效从机片选/SS/。当不用 SPI 功能时，这 4 根引脚就被配置成通用 I/O 口。

在主机模式中，主机启动逻辑从 CPU 接收一个输入（CPU 写到 SPI 数据寄存器的数据）并发出基于内部处理器时钟的 SPSCK，串行时钟也用来控制状态控制器即 8 位移位寄存器。CPU 写数据到 SPI 数据寄存器 SPDR，这个并行数据立刻被装入 8 位移位寄存器并移位输出到 MOSI 引脚，通过从属设备的输入线到从机。同时从机（或外设）数据通过 MISO 引脚输入到微控制器内部的 8 位移位寄存器，第 8 个移位传送后，数据并行传送到 SPI 接收数据寄存器，并等待 CPU 读周期的到来。一个传送结束后，SPRF 或 SPTE 状态标志被主机和外设置位。

在从机模式中，从机的启动逻辑等待/SS 引脚被拉低，然后通过 SPSCK 与主机同步。从机通过串行移位寄存器从 MOSI 线接收数据并传送到 SPI 接收数据寄存器。当写周期来到时，从机的 CPU 从内部数据总线将已经写到 SPDR 中的数据装入 8 位移位寄存器，然后等待从主机来的时钟，把数据移位送到 MISO 引脚，然后发送到主机。

**3. SPI引脚**

（1）串行数据引脚

主入从出 MISO 和主出从入 MOSI 用来串行发送和接收数据，数据格式为 MSB 在前，LSB 在后。当 SPI 被置成主机模式时，MISO 是数据输入线，MOSI 是数据输出线。在主机模式下，在 SPI 控制寄存器的第 5 位 SPMSTR 控制位被程序置成 1 之后，允许主机输出数据到 MOSI 引脚。当 SPI 被置成从机模式时，这两根线正好颠倒一下角色，即 MISO 变成从机数据输出线，而 MOSI 变成从机数据输入线。

（2）串行时钟

串行时钟 SPSCK 是数据流动的同步信号，通过 MOSI 和 MISO 对设备的数据输入、输出进行同步。SPSCK 信号只能由处于主机模式的微控制器发出，对处于从机地位的微控制器或外设，SPSCK 为输入信号。

在主机模式下，SPSCK 信号来源于内部微控制器总线时钟。当主机发起一次传送时，SPSCK 线上会自动产生 8 个时钟周期。在主机与从机之间，数据移位发生在 SPSCK 信号的一个跳变上，待数据稳定后在另外一个跳变进行采样。SPI 状态控制寄存器 SPSCR 的 SPR0 和 SPR1 两位选择 SPSCK 的时钟频率，共有 4 种选择。主机和从机之间必须遵守同样的时序模式，SPI 控制寄存器的 SPCR 中的 CPOL 和 CPHA 可以控制时序模式。

（3）外设片选

/SS 引脚的使用对于主机和从机是不同的。在从机模式下，/SS 是一次数据传送开始前允许从机 SPI 工作的片选信号；对于主机模式，一般都要将/SS 线上拉到高电平。

### 20.2.2　电能计量芯片的选择

电量监测仪功能的实现都是以测量功率、电能、电压、电流为基础的。传统的电量监测仪电量信号的采集大多采用分立器件来完成，如互感器+信号调理电路+A/D 转换器件+MCU 的方式。采样这种方式设计的电量监测仪集成度不高，可移植性比较差，同时实现高精度比较困难。而采用电子式电能表专用芯片用于电量监测仪的开发，不仅可以缩短产品开发周期，而且能大大提高产品的性能。本例中我们选用 Cirrus Logic 公司的电子式电能计量芯片 CS5460A。

## 20.3　硬件电路设计

本系统的硬件电路主要有电源电路、复位电路、仿真调试电路、电量信号采集电路、显示单元电路和单片机电路。在介绍具体的电路之前，首先介绍 CS5460A 芯片。

### 20.3.1　CS5460A 芯片的介绍

CS5460A 是美国 Cirrus Logic 公司推出的带有串行接口的单相双向功率/电能计量集成

电路芯片，它包含两个模/数转换器（ADC）、高速电能计算功能和一个串行接口的高度集成的模/数转换器，还具有方便的片上 AC/DC 系统校准功能。该芯片的性能优于其他计量芯片，主要体现在：

- 可以测量瞬时电流、瞬时电压、瞬时功率、能量、电流有效值和电压有效值。
- 具有片内看门狗定时器与内部电源监视器。
- 集成有一个简单的三线串行接口，该接口与 SPI 和 Microwire 标准兼容，方便与外部微控制器通信。
- 具有机械计度器/步进电机驱动器。
- 外部时钟最高频率可达 20MHz。
- 具有与能量成正比的频率可编程的脉冲输出，有功率方向输出指示。

为了便于进行硬件电路设计，下面给出该芯片的管脚图，如图 20-3 所示。

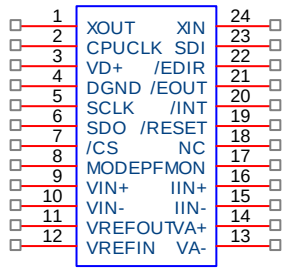



图 20-3　CS5460A 芯片管脚图

由图 20-3 可以看出，该芯片有 24 个管脚，搭配简单的外围电路，就可以实现电量信号的采集，下面对具体的管脚进行介绍。

- XOUT 和 XIN：晶振输出，晶振输入。芯片内的一个选通与这些管脚相连，可以和晶振一起使用为驱动器提供系统时钟。另外，一个外部时钟（与 CMOS 时钟兼容）也可以与管脚 XIN 一起为驱动器提供系统时钟。
- CPUCLK：CPU 时钟输出。片上振荡器输出可以驱动一个标准 CMOS 负荷。
- /CS：芯片选择：当低态有效时，端口可以识别 SCLK。该管脚高态有效时 SDO 管脚处于高阻状态。当 SCLK 处于低电平时，/CS 应改变。
- SDI：SDI 是串行数据端口的输入管脚。数据的输入速率由 SCLK 决定。
- SDO：SDO 是串行数据端口的输出管脚。当/CS 高时，其输出将处于高阻抗状态。
- SCLK：串行时钟输入管脚。该管脚的时钟信号分别决定 SDI 和 SDO 管脚的数据输入和输出速率。这一输入是一个允许低时间信号的施密特触发器，只有当/CS 低时，SCLK 管脚才能识别时钟。
- /INT：中断管脚。当/INT 处于低值时，标志着外部结果寄存器已经更新。可以通过向 CS5460A 写入适当命令来使/INT 清空。
- /EOUT：能量输出管脚。输出一个脉冲宽固定、频率（可编程）和电能成比例的脉冲串。
- /EDIR：能量方向指示器。如果测量到的能量是负值，能量方向指示器发出指示。
- IIN+，IIN-：差分电流输入管脚。电流通道的差分模拟输入管脚。
- VIN+，VIN-：差分电压输入管脚。电压通道的差分模拟输入管脚。
- VREFOUT：电压参考输出管脚。芯片上的参考电压是该管脚的输出。电压参考的名义值为 2.5V，其值是参考转换器上的 AGND 管脚。

- VREFIN：电压参考值输入管脚。该管脚的电压输入给芯片上的调制器建立了电压参考。
- VA+：正极模拟电源。与 AGND 相比，正极模拟电源通常为+5V±10%。
- AGND：模拟接地管脚。模拟管脚接地必须是在最低电势。
- VD+：正极数字电源管脚。与 DGND 相比，正极数字供电源常为+5V±10%。
- DGND：数字接地管脚。数字接地与 AGND 具有相同的电平。
- PFMON：掉电监视管脚。该管脚监视模拟电源。
- /RESET：复位管脚。当复位在低电平时，所有内部寄存器都被设置为其默认值。
- NC：不连接。

通过对串行电能计量芯片 CS5460A 的介绍，读者应该对电能计量电路有了基本的认识，下面介绍具体的电量信号采集电路。

## 20.3.2　电量信号采集电路

电量信号采集电路以 CS5460A 为中心，搭以一定的外围电路来实现。由于电流通道输入有效值范围为 0~30mV 或 0~150mV，电压通道输入有效值范围为 0~150mV，所以电压、电流采样电路必须将强电信号转换为弱电信号供后级使用，其主要使用电流互感器、分压器、阻容电路等构成，系统的精度、准确度与此有很大关系。因此，在器件的选择上要求其稳定性好、精度高、误差小、寿命长。

CS5460A 与单片机 MC9S08JM60 连接是通过其内部集成的一个简单的三线串行接口实现的，该接口与 SPITM 标准兼容。如图 20-4 所示为电量信号采集电路原理图。

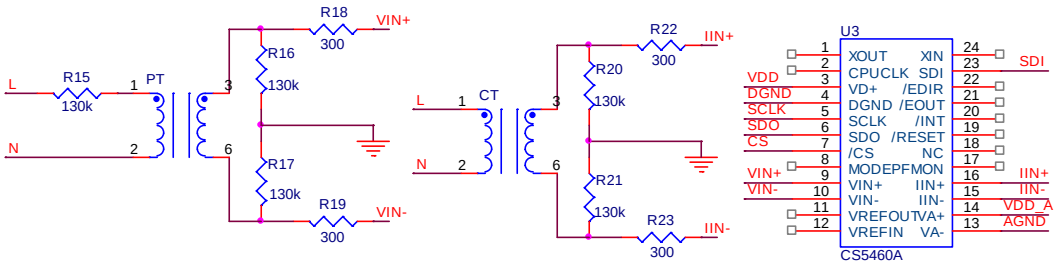



图 20-4　电量信号采集电路原理图

如图 20-4 所示，CS5460A 的串行接口由串行时钟（SCLK）、串行数据输入（SDI）、串行数据输出（SDO）和片选 CS 组成。CS 是芯片控制选择端，当为低电平时，端口可以识别 SCLK，SDI、SDO 是串行数据输入、输出端，CS5460A 中的数据及命令字的输入和输出都是通过这两个管脚实现的，当 SCLK 为上升沿时在 SDI 上的数据可以写入，SDO 上的数据可以被读出。本设计直接使用串行口读取能量寄存器，而没有使用/EOUT 和/EDIR 引脚提供的一个能量积累代数值的接口。被测量的电压和电流信号通过变压和变流器接入 IIN+、IIN-、UIN+和 UIN-管脚进行模/数转换。模/数转换的基准电源输入由 UREFIN 给出。

### 20.3.3 显示单元电路

显示单元主要显示最终的测量结果，在此我们选用 128×64 的点阵图形液晶模块，LCM 的控制芯片为 KS0108。显示单元的电路原理图如图 20-5 所示。

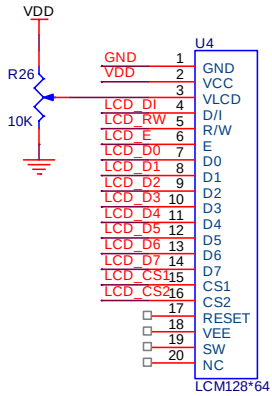



图 20-5　显示单元电路原理图

由图 20-5 所示，LCM 的数据线接 MC9S08JM60 单片机的 PTD 口，控制线 E 接单片机的 PTB0 管脚，控制线 D/I 接单片机的 PTB1 管脚，控制线 R/W 接单片机的 PTB2 管脚，控制线 CS1 接单片机的 PTB3 管脚，控制线 CS2 接单片机的 PTB4 管脚。

### 20.3.4 单片机系统

单片机选用 Freescale 公司的 MC9S08JM60，由于该单片机自带了 SPI 模块，因此在 SPI 设计时非常简单。如图 20-6 所示为具体的单片机系统电路图。

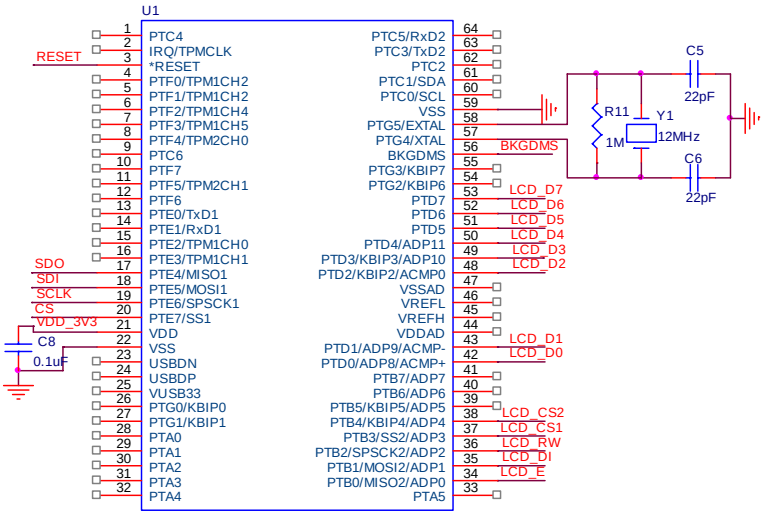



图 20-6　单片机系统电路图

由图 20-6 可以看出，单片机与电量采集电路的 CS5460A 通过 SPI 接口连接；与 LCM 模块的连接主要是通过 I/O 口，LCM 的数据线接 MC9S08JM60 的 PTD 口，控制线分别接在管脚 PTB0~PTB4 上。考虑到电源的输入纹波对单片机的影响，在电源的管脚上增加一个 104 的滤波电容，以减少干扰。

# 20.4　软件设计与代码

## 20.4.1　软件设计流程

电量监测仪系统软件主要包括 SPI 的实现、CS5460A 操作程序的实现、LCD 显示程序和主程序。电量监测仪的主程序流程图如图 20-7 所示。

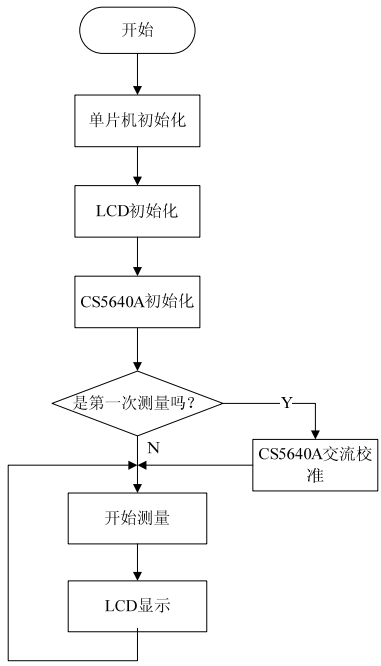



图 20-7　电量监测仪主程序流程图

MC9S08JM60 上电复位，完成单片机和 LCD 初始化后，首先就对 CS5460A 进行复位、初始化，然后发测量命令，处于自动测量状态，显示测量结果。程序最后处理结果显示在 LCD 上。如果是第一次上电测量，必须进行交流偏移校准、交流增益校准子程序。

下面对电量监测仪系统软件的各个部分进行具体介绍，在介绍具体程序之前，先对 CS5460A 的操作进行介绍。

## 20.4.2　CS5460A 的操作

CS5460A 提供 SPI 串行通信接口和一系列寄存器，单片机可以通过 SPI 接口读写这些

寄存器，以实现对 CS5460A 的控制和从 CS5460A 读取指定的数据。对 CS5460A 的操作是通过向其传输命令字来实现的， CS5460A 提供了寄存器的读/写和校准控制等在内的 7 个操作命令，所有的命令长度均为 1 个字节（8 位）。命令状态机在 SCLK 的上升沿解释 8 位命令字，它将命令字解释为公认的标准，同时为数据的传输作好准备。本文仅以寄存器读/写命令为例加以简要说明，详细内容请参考 CS5460A 用户手册。

寄存器读/写命令如图 20-8 所示：

| B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
|---|---|---|---|---|---|---|---|
| 0 | W/R | RA4 | RA3 | RA2 | RA1 | RA0 | 0 |

图 20-8　寄存器读/写命令

这个命令通知状态机需要对寄存器进行访问，在 8 个 SCLK 时钟周期内，地址寄存器的读取被加载到输出缓冲区，在第 24 个 SCLK 时，写入数据被传输到输入缓冲区。

其中：W/R 为写入/读取控制位。

在对 CS5460A 的操作进行简单介绍后，下面将介绍程序的具体实现。

### 20.4.3　SPI 的实现

SPI 程序主要实现数据传输。对 CS5460A 的操作都是以该部分程序为基础的。由于本系统使用的单片机（MC9S08JM60）本身带有 SPI 模块，因此 SPI 程序实现起来非常简单，主要包括初始化和数据传输两个部分，下面介绍其具体的程序。

**1. 初始化**

该部分程序主要设置 SPI 模块的相关寄存器，下面为具体的程序。

```
void Master_SPI_Init(void)
{
  SerFlag = 0;
  ErrFlag = 0;
  Master_SPI_InpLen = 0;              //接收缓冲器数据个数清零
  InpIndxW = InpIndxR = 0;            //接收索引变量清零
  Master_SPI_OutLen = 0;              //发送缓冲器数据个数为零
  OutIndxW = OutIndxR = 0;            //发送索引变量为零
  #pragma MESSAGE DISABLE C4002       //禁止警告
  (void)SPI1S;                        //读状态寄存器
  (void)SPI1DL;                       //读设备寄存器
  setReg8(SPI1BR, 0x21);              //波特率寄存器设置
  setReg8(SPI1C2, 0x00);              //SPI 端口控制寄存器 2 设置
  setReg8(SPI1C1, 0x95);              //SPI 端口控制寄存器 1 设置
  SPI1C1_SPE = 1;                     //使能 SPI 设备
}
```

**2. 数据传输**

该部分程序主要实现数据的接收和发送，下面为具体的程序。

```
byte Master_SPI_RecvChar(Master_SPI_TComData *Chr)
{
  byte FlagTmp;

  if (Master_SPI_InpLen > 0)         //接收到的字符个数大于 0?
  {
    EnterCritical();                //关中断
    Master_SPI_InpLen--;             //接收索引递减
    *Chr = InpBuffer[InpIndxR];       //读数据
    InpIndxR = (byte)((InpIndxR+1) & (Master_SPI_INP_BUF_SIZE-1));
    FlagTmp = SerFlag;               //保存标志
    SerFlag &= ~(OVERRUN_ERR | FULL_RX); //清标志位
    ExitCritical();                   //开中断
  }
  else {
    return ERR_RXEMPTY;               //接收器为空
  }
  if ((FlagTmp & (OVERRUN_ERR | FULL_RX)) != 0) //发生溢出错误?
  {
    return ERR_OVERRUN;
  }
  else {
    return ERR_OK;
  }
}

byte Master_SPI_SendChar(Master_SPI_TComData Chr)
{
  EnterCritical();                  //关中断
  if (Master_SPI_OutLen == Master_SPI_OUT_BUF_SIZE)
  {
    ExitCritical();                 //开中断
    return ERR_TXFULL;
  }
  Master_SPI_OutLen++;              //递增发送索引
  OutBuffer[OutIndxW] = Chr;        //数据装入缓冲器
  OutIndxW = (byte)((OutIndxW+1) & (Master_SPI_OUT_BUF_SIZE-1));
  if(!(SerFlag & RUNINT_FROM_TX))   //正在发送?
  {
    (void)SPI1S;                    //读状态寄存器
    SPI1DL = OutBuffer[OutIndxR];   //数据装入发送寄存器
```

```
    SerFlag |= RUNINT_FROM_TX;      //置标志位
  }
  ExitCritical();                    //开中断
  return ERR_OK;
}
```

### 20.4.4 CS5460A 操作程序的实现

根据前面的介绍可知，CS5460A 的操作首先需要对 CS5460A 进行初始化，然后可以实现读/写操作，其中 CS5460A 的初始化是关键。CS5460A 的初始化操作主要是按照一定的步骤完成其内部寄存器的设置，最后完成内部校正功能。下面介绍具体的初始化程序。

```
void Init5460(void)
{
   delay_nms(5);
   buf[0]=0xff;           //SYNC1
   buf[1]=0xff;           //SYNC1
   buf[2]=0xff;           //SYNC1
   buf[3]=0xfe;           //SYNC0
   Write_CS5460A(buf,4); //写 3 个同步命令 1 后再写 1 个同步命令 0
   buf[0]=0x40;           //写配置寄存器
   buf[1]=0x00;           //GI=1
   buf[2]=0x10;
   buf[3]=0x61;           //DCLK=MCLK/1,高通
   Write_CS5460A(buf,4);   //写配置寄存器
  /*************************************************/
   buf[0]=WRITE|0x20;
   buf[1]=0x3D;
   buf[2]=0xC2;
   buf[3]=0x5C;
   Write_CS5460A(buf,4);   //电流 交流偏置 校准寄存器

   buf[0]=WRITE|0x22;
   buf[1]=0x1F;
   buf[2]=0xBB;
   buf[3]=0x65;
   Write_CS5460A(buf,4);   //电压 交流偏置 校准寄存器
  /*************************************************/
   buf[0]=WRITE|0x04;
   buf[1]=0x29;
   buf[2]=0x71;
   buf[3]=0xCA;
   Write_CS5460A(buf,4);     //电流增益校准寄存器**

   buf[0]=WRITE|0x08;
```

```
 buf[1]=0x40;
 buf[2]=0x96;
 buf[3]=0xD3;
 Write_CS5460A(buf,4);     //电压增益校准寄存器**

 buf[0]=0x5e;
 buf[1]=0xF0;
 buf[2]=0x00;
 buf[3]=0x00;
Write_CS5460A(buf,4);      //清除状态寄存器

 buf[0]=0x74;
 buf[1]=0x80;              //DRDY
 buf[2]=0x00;
 buf[3]=0x00;
 Write_CS5460A(buf,4);    //写中断屏蔽寄存器

 buf[0]=0x78;
 buf[1]=0x00;
 buf[2]=0x00;
 buf[3]=0x00;               //默认值
 Write_CS5460A(buf,4);      //写控制寄存器

 buf[0]=0x4c;
 buf[1]=0x00;
 buf[2]=0x34;
 buf[3]=0x9C;
 Write_CS5460A(buf,4);       //写 EOUT 脉冲输出寄存器

 buf[0]=0x4A;
 buf[1]=0x00;
 buf[2]=0x01;               //每秒钟计算 20 次，N=400
 buf[3]=0x90;
 Write_CS5460A(buf,4);       //写 CYCLE COUNT 寄存器

 Read_CS5460A(0x1e,buf);     //读状态寄存器
 buf[0]=0X5E;
 Write_CS5460A(buf,4);        //写状态寄存器

 buf[0]=0xe8;
 Write_CS5460A(buf,4);        //启动 CS5460A

/*--------校准过程----------------------------
---清除 DTDY，发送 Power_halt 命令---------------*/
 Clear_DRDY() ;
```

```
    Wait_DRDY_High() ;
    Clear_DRDY();

    buf[0]=0xa0;
    Write_CS5460A(buf,1);      //发送 POWER_HALT 命令

     /*=============================================
     第一阶段：交流偏置校准，方法：将电流电压输入短接，
     读偏置寄存器。
     ---------------------------------------------*/
    //电流  交流偏置校准：
    Clear_DRDY();

    buf[0]=0xcd ;
    Write_CS5460A(buf,1);      //写校准命令寄存器
    Wait_DRDY_High();          //等待校准完成

     Read_CS5460A(0x20,buf);   //读 Iacoff 寄存器
     SetRowCol(1,0);           //第一行第一列
     disp_ch(buf[0]);
     SetRowCol(1,24);
     disp_ch(buf[1]);
     SetRowCol(1,48);
     disp_ch(buf[2]);
     SetRowCol(1,72);
     disp_ch(buf[3]);

    //电压  交流偏置校准：
    Clear_DRDY();

     buf[0]=0xd5 ;
     Write_CS5460A(buf,1);     //写校准命令寄存器
     Wait_DRDY_High();         //等待校准完成

      Read_CS5460A(0x22,buf); //读电压偏置寄存器
      SetRowCol(2,0);          //第一行第一列
      disp_ch(buf[0]);
    SetRowCol(2,24);
      disp_ch(buf[1]);
      SetRowCol(2,48);
      disp_ch(buf[2]);
    SetRowCol(2,72);
      disp_ch(buf[3]);

  /*=================================================
```

```
    第二阶段：校准交流增益寄存器。方法：将电压和电流输入加
    至满幅，读取增益寄存器的值。
    ------------------------------------------------------*/
      //电流  交流增益校准：
      Clear_DRDY();
      buf[0]=0xce;
      Write_CS5460A(buf,1);   //写校准命令寄存器
      Wait_DRDY_High();       //等待校准完成

      Read_CS5460A(0x04,buf);
      SetRowCol(1,0);        //第一行第一列
      disp_ch(buf[0]);
      SetRowCol(1,24);
      disp_ch(buf[1]);
      SetRowCol(1,48);
      disp_ch(buf[2]);
      SetRowCol(1,72);
      disp_ch(buf[3]);

      //电压  交流增益校准
      Clear_DRDY();
      buf[0]=0xd6;
      Write_CS5460A(buf,1);; //写校准命令寄存器
      Wait_DRDY_High();      //等待校准完成

      Read_CS5460A(0x08,buf);
      SetRowCol(2,0);        //第一行第一列
      disp_ch(buf[0]);
      SetRowCol(2,24);
      disp_ch(buf[1]);
      SetRowCol(2,48);
      disp_ch(buf[2]);
      SetRowCol(2,72);
      disp_ch(buf[3]);
    }
```

## 20.4.5　主程序

单片机初始化后，接着初始化 LCD 和 CS5460A，最后进入主程序工作。主程序调用前面介绍的程序，不断刷新电压、电流值，并将结果显示在 LCD 上。在具体应用时，读者可以根据自己系统的需要来设计相应的程序。下面给出电量监测仪的主程序。

```
    void main(void)
    {
      dword VRMS=0,VRMSNew=0;
```

```
    dword IRMS=0,IRMSNew=0;
    PE_low_level_init();          //单片机初始化
    init_lcd();                   //LCD 初始化
    Init5460();
    SetRowCol(1,0);
    disp_str("VRMS=");
    SetRowCol(2,0);
    disp_str("IRMS=");

     SetRowCol(1,70);
     disp_str("V");
     SetRowCol(2,70);
     disp_str("A");

   for(;;) {
       VRMSNew = filter(CS_VRMS);
         if( (VRMSNew-VRMS>3) || (VRMS-VRMSNew>3) )
         {
             VRMS = VRMSNew;
         }

         IRMSNew = filter(CS_IRMS);
         if( (IRMSNew-IRMS>3)||(IRMS-IRMSNew>3) )
         {
             IRMS=IRMSNew;
         }
         SetRowCol(1,30);
        disp_ch(VRMS/100);       //显示电压值
        SetRowCol(2,30);
        disp_ch(IRMS/1000);      //显示电流值
         delay_nms(50);
    }
}
```

## 20.5 实例总结

本章详细介绍了基于 MC9S08JM60 单片机的 SPI 模块实现的电量监测仪。电量信号采集采用 Cirrus Logic 公司的电子式电能计量芯片 CS5460A。由于 CS5460A 的优良特性，保证了系统较高的可靠性和精度，而且电路结构简单。本例首先对 SPI 协议进行了介绍，然后给出了具体的硬件设计原理图，最后介绍了系统的软件设计。通过本章的介绍，读者对 SPI 协议、电能计量芯片有了更深入的了解，为设计自己的系统提供了很好的参考。读者可在本例的基础上扩展自己的需求，从而实现功能更强的系统。

# 第 21 章

# SPI 总线通信模块：SD 卡数据存储系统设计实例

继续第 20 章的主题，本章将介绍另外一个 SPI 总线通信模块的应用实例：基于 MC9S08JM60 单片机 SPI 模块实现的 SD 卡数据存储系统。首先进行实例描述，然后再介绍具体的设计思路和软硬件设计。

## 21.1 实例内容说明

SPI 总线是 Motorola 公司提出的一种同步串行外围总线系统，它可以使微处理器与各种外围设备以同步串行方式进行信息交换。采用 SPI 总线，可以很方便地与很多厂家生产的标准外围器件直接接口。这类器件很多，既可以是简单的 TTL 移位寄存器，也可以是复杂的 LCD 显示驱动器或 A/D 转换子系统。基于 SPI 总线的 SD 卡数据存储系统结构简单、程序编写方便，易于实现系统软硬件的模块化和标准化。如图 21-1 所示为基于 SPI 模块实现的 SD 卡数据存储系统的原理框图。

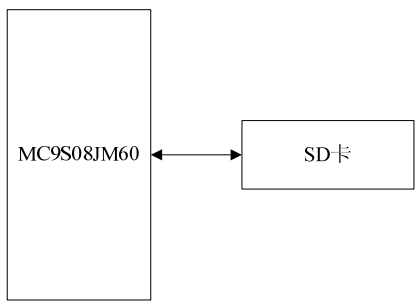



图 21-1 基于 SPI 模块实现的数据采集系统的原理框图

由图 21-1 可以看出，基于 SPI 模块实现的 SD 卡数据存储系统结构简单，扩展非常容易。

## 21.2 设计思路分析

基于 SPI 模块实现的数据采集系统的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、$I^2C$ 控制器、定时器等多种外设，方便进行系统设计。

根据图 21-1 所确定的系统原理框图可知，关键在于 SPI 协议的实现，关于 SPI 通信协议的相关知识请参考第 20 章。除此之外，读者还需了解 SD 卡的相关知识，下面进行介绍。

SD 卡是一种大容量、性价比高、体积小、访问接口简单的存储卡。SD 卡大量应用于数码相机、MP3、手机、大容量存储设备，作为这些便携式设备的存储载体，它还具有低功耗、非易失性、保存数据无需消耗能量等特点。

**1. SD卡的外部物理接口**

SD 卡的外形和接口触点如图 21-2 所示。SD 卡的外形尺寸为 24mm×32mm×2.1mm（普通）或 24mm×32mm×1.4mm（薄 SD 存储卡）。

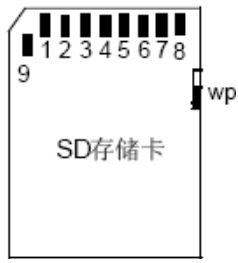



图 21-2　SD 卡的外形和接口图

如图 21-2 所示，SD 卡有 9 个触点，各触点的名称及作用如表 21-1 所示。

表 21-1　SD 卡触点名称及作用

| 引脚 | SD 模式 | | | SPI 模式 | | |
|---|---|---|---|---|---|---|
| | 名称 | 类型 | 描述 | 名称 | 类型 | 描述 |
| 1 | CD/DAT32 | I/O/PP3 | 卡的检测/数据线[Bit 3] | CS | I | 片选（低电平有效） |
| 2 | CMD | PP4 | 命令/响应 | DI | I5 | 数据输入 |
| 3 | VSS1 | S | 电源地 | VSS | S | 电源地 |
| 4 | VDD | S | 电源 | VDD | S | 电源 |
| 5 | CLK | I | 时钟 | SCLK | I | 时钟 |
| 6 | VSS2 | S | 电源地 | VSS2 | S | 电源地 |
| 7 | DAT0 | I/O/PP | 数据线[Bit 0] | D0 | O/PP | 数据输出 |
| 8 | DAT1 | I/O/PP | 数据线[Bit 1] | RSV | | |
| 9 | DAT2 | I/O/PP | 数据线[Bit 2] | RSV | | |

SD 存储卡的通信基于一个高级的 9 引脚接口（时钟、命令、4 条数据线和 3 条电源线），可以在最高 25MHz 频率和低电压范围内工作。由表 21-1 可见，SD 卡在不同的通信模式下，各引脚的功能也不相同。这里的通信模式是指微控制器（主机）访问 SD 卡时使用的通信协议，分为两种：SD 模式及 SPI 模式。

**2. SD卡的工作模式**

在具体通信过程中，主机只能选择其中一种通信模式。通信模式的选择对于主机来说是透明的。卡将会自动检测复位命令的模式（即自动检测复位命令使用的协议），而且要求以后双方的通信都按相同的通信模式进行。所以，在只使用一种通信模式的时候，无需明白另一种模式。下面简单介绍这两种工作模式。

（1）SD 模式

在 SD 模式下，主机使用 SD 总线访问 SD 卡，其拓扑结构如图 21-3 所示。由图可见，SD 总线上不仅可以挂接 SD 卡，还可以挂接 MMC 卡。

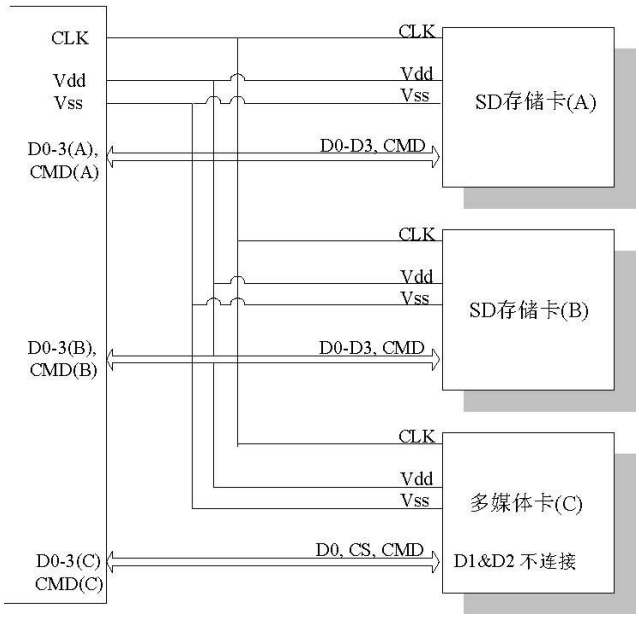



图 21-3　SD 存储卡系统（SD 模式）的总线拓扑结构

由上图可见，SD 存储卡系统（SD 模式）的总线拓扑结构为：一个主机（如微控制器）、多个从机（卡）和同步的星形拓扑结构。所有卡公用时钟 CLK、电源和地信号。而命令线（CMD）和数据线（DATA0～DATA3）则是卡的专用线，即每张卡都独立拥有这些信号线。

（2）SPI 模式

在 SPI 模式下，主机使用 SPI 总线访问卡。当今大部分微控制器本身都带有硬件 SPI 接口，所以使用微控制器的 SPI 接口访问卡是很方便的。微控制器在卡上电后的第 1 个复

位命令就可以选择卡进入 SPI 模式或 SD 模式，但在卡上电期间，它们之间的通信模式不能更改为 SD 模式。卡的 SPI 接口与大多数微控制器的 SPI 接口兼容。如图 21-4 所示为在 SPI 模式下，主机使用 SPI 总线访问 SD 卡的拓扑结构图。

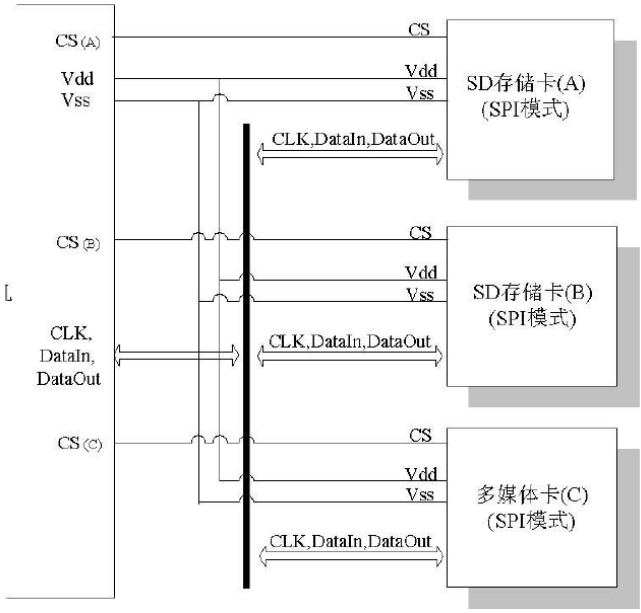



图 21-4　SD 存储系统（SPI 模式）的总线拓扑结构

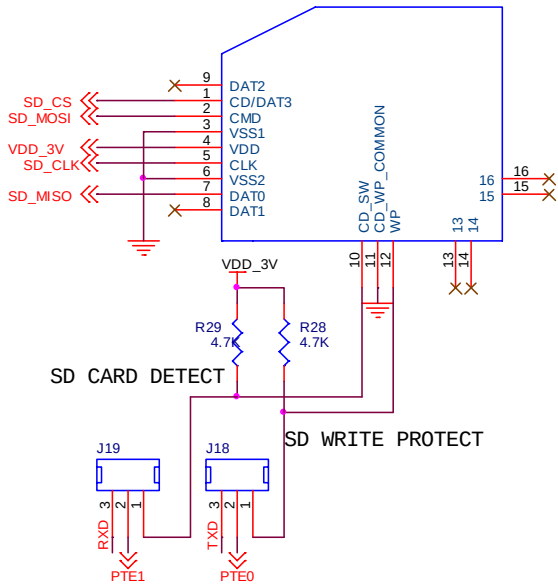
CS 信号在 SPI 处理（命令、响应和数据）期间必须保持有效（低电平）。唯一例外的情况是在对卡编程的过程。在这个过程中，主机可以使 CS 信号为高电平，但不影响卡的编程。由图 21-4 还可见，当 SPI 总线上挂接 *N* 张卡时，需要 *N* 条 CS 片选线。

## 21.3　硬件电路设计

本系统的硬件电路相对简单，主要有电源电路、复位电路、仿真调试电路、SD 卡接口电路和单片机电路。下面重点介绍数据 SD 卡接口电路和单片机电路的设计。

### 21.3.1　SD 卡接口电路

SD 卡接口电路图如图 21-5 所示。卡座的引脚共有 16 个，SD 卡通过 SD 卡座和微控制器接口，数据传输采用 SPI 方式，供电电源为 3.3V，引脚 CD_SW 用于 SD 卡插入检测，由单片机 MC9S08JM60 的 PTE1 口控制。引脚 WP 用于 SD 卡写保护，由单片机 MC9S08JM60 的 PTE0 口控制。




21-5　SD 卡接口电路

## 21.3.2　单片机系统电路

单片机选用 Freescale 公司的 MC9S08JM60，由于该单片机自带 SPI 模块，因此在硬件设计时非常简单，只需将 PTE4～PTE6 管脚设置成 SPI 模块功能管脚即可。PTE0 和 PTE1 作为普通 I/O 口，用以控制 SD 卡的插入检测和 SD 卡写保护。如图 21-6 所示为具体的单片机系统电路图。

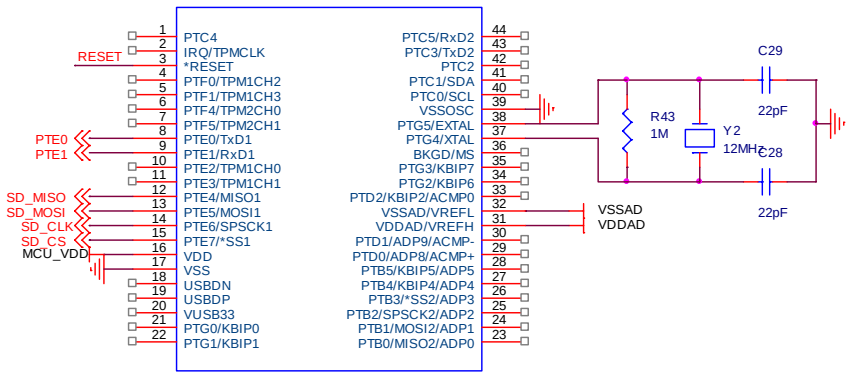



图 21-6　单片机系统电路图

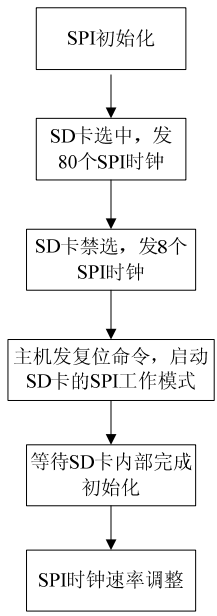
由图 21-6 可以看出，该单片机电路非常简单。单片机的电源电路、仿真调试电路等可参考第 5 章的相关部分，在此不作介绍。考虑到电源的输入纹波对单片机的影响，在电源

的管脚上增加一个 104 的滤波电容，以减少干扰。

# 21.4 软件设计与代码

## 21.4.1 软件设计流程

基于 SPI 总线的数据存储系统的软件设计主要包括两个部分：SD 卡读/写子程序和测试程序。SD 卡读/写子程序包括 SD 卡数据读/写子程序和 SD 卡信息读取程序。SD 卡在进行读/写操作之前，必须进行特定的初始化操作，以使 SD 卡进入 SPI 工作模式。SD 卡的初始化流程图如图 21-7 所示。




21-7 SD 卡初始化流程图

下面分别对每个部分进行具体的介绍。

## 21.4.2 SD 卡读/写子程序

SD 卡由不同的数据块组成，读/写操作的基本单位是块。即使应用程序只需读/写一个字节，则整个块数据也必须被读/写。SD 卡的读写操作基于命令—应答方式，支持的标准命令如下所示。

```
enum{
  SD_CMD0,  /* 复位 SD 存储卡 */
  SD_CMD1,  /* 向主机发送容量支持信息，并激活 SD 卡的初始化过程 */
  SD_CMD2,
```

```
    SD_CMD3,
    SD_CMD4,
    SD_CMD5,
    SD_CMD6,  /* Checks switchable function (mode 0) and switches card function
(mode 1).*/
    SD_CMD7,
    SD_CMD8,  /* 发送 SD 存储卡接口条件信息（主供电电源范围等） */
    SD_CMD9,  /* 请求被选中的卡发送其卡特定数据 */
    SD_CMD10, /* 请求被选中的卡发送其卡识别信息 */
    SD_CMD11,
    SD_CMD12, /* 在多块数据读操作过程中，强制卡停止数据传输 */
    SD_CMD13, /* 请求被选中的卡发送其状态寄存器信息 */
    SD_CMD14,
    SD_CMD15,
    SD_CMD16, /* 设备标准容量卡的每个读/写块的大小 */
    SD_CMD17, /* 读取由 SET_BLOCKLEN 命令指定的一块数据 */
    SD_CMD18, /* 连续由 SD 卡向主机传输数据，除非发生 STOP_TRANSMISSION 命令中断*/
    SD_CMD19,
    SD_CMD20,
    SD_CMD21,
    SD_CMD22,
    SD_CMD23,
    SD_CMD24, /* 向由 SET_BLOCKLEN 命令指定的地址空间内写数据 */
    SD_CMD25, /* 向块连续写数据*/
    SD_CMD26,
    SD_CMD27, /* 编程 CSD 寄存器的可编程位 */
    SD_CMD28, /* 如果 SD 卡有写保护，则该命令设置写保护组的写保护位 */
    SD_CMD29, /*如果 SD 卡有写保护，则该命令清除写保护组的写保护位 */
    SD_CMD30, /* 如果 SD 卡有写保护，则该命令请求卡发送写保护位的状态 */
    SD_CMD31,
    SD_CMD32, /* 设置将被擦除的第一个写数据块的地址 */
    SD_CMD33, /* 设置将被擦除的最后一个写数据块的地址 */
    SD_CMD34,
    SD_CMD35,
    SD_CMD36,
    SD_CMD37,
    SD_CMD38, /* 擦除先前选中的所有写数据块 */
    SD_CMD39,
    SD_CMD40,
    SD_CMD41,
    SD_CMD42, /* 用以设置或清零密码，锁定或解锁 SD 卡 */
    SD_CMD43,
    SD_CMD44,
    SD_CMD45,
```

```
  SD_CMD46,
  SD_CMD47,
  SD_CMD48,
  SD_CMD49,
  SD_CMD50,
  SD_CMD51,
  SD_CMD52,
  SD_CMD53,
  SD_CMD54,
  SD_CMD55, /* 告诉 SD 卡下一个命令是具体应用命令，而不是标准命令 */
  SD_CMD56, /* 用于传输一个块数据到 SD 卡或从 SD 卡读取一个块数据 */
  SD_CMD57,
  SD_CMD58, /* 读 SD 卡的 OCR 寄存器 */
  SD_CMD59, /*打开或关闭 CRC 选项*/
  SD_CMD60,
  SD_CMD61,
  SD_CMD62,
  SD_CMD63
};
```

**1. SD卡数据读子程序**

SD 卡读子程序一次完成 SD 卡一个块数据的读操作，如果读操作出错，则返回异常码。具体的程序如下。

```
UINT8 SD_Read_Block(UINT32 u16SD_Block,UINT8 *pu8DataPointer)
{
    UINT8 u8Temp=0;
    UINT16 u16Counter;

    SPI_SS=ENABLE;

    gu8SD_Argument.lword=u16SD_Block;
    gu8SD_Argument.lword=gu8SD_Argument.lword<< SD_BLOCK_SHIFT;

    if(SD_SendCommand(SD_CMD17|0x40,SD_OK))
    {
        SPI_SS=DISABLE;
        return(4);      // Command IDLE fail
    }

    while(u8Temp!=0xFE)
        u8Temp=SPI_Receive_byte();

    for(u16Counter=0;u16Counter<BLOCK_SIZE;u16Counter++)
        *pu8DataPointer++=SPI_Receive_byte();
```

```
    (void)SPI_Receive_byte();  // Dummy SPI cycle
    (void)SPI_Receive_byte();  // Dummy SPI cycle

    SPI_SS=DISABLE;

    (void)SPI_Receive_byte();  // Dummy SPI cycle

    return(0);
}
```

**2. SD卡数据写子程序**

SD 卡写子程序一次完成对 SD 卡一个块数据的写操作，如果写操作出错，则返回异常码。具体的程序如下。

```
UINT8 SD_Write_Block(UINT32 u16SD_Block,UINT8 *pu8DataPointer)
{
    UINT16 u16Counter;

    SPI_SS=ENABLE;

    gu8SD_Argument.lword=u16SD_Block;
    gu8SD_Argument.lword=gu8SD_Argument.lword<< SD_BLOCK_SHIFT;

    if(SD_SendCommand(SD_CMD24|0x40,SD_OK))
    {
        SPI_SS=DISABLE;
        return(2);     // Command IDLE fail
    }

    SPI_Send_byte(0xFE);

    for(u16Counter=0;u16Counter<BLOCK_SIZE;u16Counter++)
        SPI_Send_byte(*pu8DataPointer++);

    SPI_Send_byte(0xFF);    // checksum Bytes not needed
    SPI_Send_byte(0xFF);


    if((SPI_Receive_byte() & 0x0F) != 0x05)
    {
        SPI_SS=DISABLE;
        return(3);     // Command fail
    }
```

```
    while(SPI_Receive_byte()==0x00);  // Dummy SPI cycle

    SPI_SS=DISABLE;
    return(0);
}
```

**3. SD卡信息读取子程序**

SD 卡信息读取子程序包括读取 SD 卡内部的 CSD 寄存器，以获取 SD 卡的存储信息，如版本号、生产厂家等。具体的程序如下。

```
byte SD_ReadCSD(void)
{
    byte i;

    SPI_SS=ENABLE;

    if(SD_SendCommand(SD_CMD9|0x40,SD_OK))
    {
        SPI_SS=DISABLE;
        return(4);
    }

    while(i!=0xFE)
        i=SPI_Receive_byte();

    for(i=0;i<16;i++)
        vSD_CSD[i]=SPI_Receive_byte();

    (void)SPI_Receive_byte();
    (void)SPI_Receive_byte();

    SPI_SS=DISABLE;

    (void)SPI_Receive_byte();

    return(0);
}
```

## 21.4.3 主程序

系统经过一系列初始化后，即进入主程序工作，主程序调用前面介绍的程序，完成 SD 卡的识别、读/写等功能。在具体应用时，读者可以根据自己系统的需要来设计相应的程序。下面给出基于 SPI 总线的数据存储系统的主程序。

```
void main(void)
{
    SOPT2 = 0x00;

    _SD_DE = 0;                  // Card detection
    _SD_WR = 0;                  // write portect

    while(SD_DE == kSD_Desert);    // SD Card inserted
     SCSI_Init();                 // Mass Storage Device Initialization
     EnableInterrupts;
     for(;;)
    {
        if(vCBWBuf_flag==1)
        {
            SCSI_Process();        // SCSI Process
            vCBWBuf_flag=0;
        }
    }
}
```

## 21.5　实例总结

本章详细介绍了基于 MC9S08JM60 单片机的 SPI 总线实现的数据存储系统设计。利用这种方式实现的数据存储系统在硬件上非常简单，程序易于移植，从而便于实现系统软/硬件的模块化和标准化。本例实现的重点在于了解 SD 卡的接口及其工作模式。

# 第 22 章

# USB 串行总线模块：数据传输系统设计实例

USB（Universal Serial BUS）的中文含义是“通用串行总线”，它的主要特点是拥有即插即用、传输速度快、有广泛的软硬件支持、有完备的总线拓扑结构、支持多种设备等优点，可以连接鼠标、键盘、打印机、扫描仪、摄像头、闪存盘、MP3、手机、数码相机等几乎所有的计算机外部设备，在现代的电子系统中应用十分广泛。本章将介绍基于 MC9S08JM60 单片机 USB 模块实现的数据传输系统的设计过程.

## 22.1 实例内容说明

USB 总线是康柏、微软、IBM、DEC 等公司为解决传统总线不足而于 1995 年推出的一种新型的通信标准。该总线接口具有使用方便、速度快、连接灵活、独立供电、易于扩展等优点，已逐渐成为现代数据传输的发展趋势。基于 USB 总线的数据传输系统结构简单、程序编写方便，易于实现系统软硬件的模块化和标准化。如图 22-1 所示为基于 USB 模块实现的 USB 数据传输系统的原理框图。

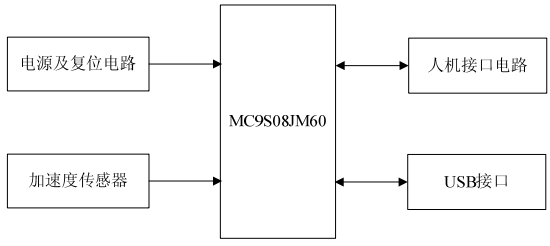



图 22-1 基于 USB 模块实现的数据传输系统的原理框图

## 22.2　设计思路分析

基于 USB 模块实现的数据传输系统的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 的片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、$I^2C$ 控制器、定时器等多种外设，方便进行系统设计。

按照图 22-1 所确定的系统原理框图，关键在于 USB 协议的实现，下面介绍 USB 总线的相关知识。

**1. USB的基本概念**

USB 是一种主从的总线结构。在 USB 系统中只能有一个 USB 主机（host），可以有多个设备（Device）。下面介绍 USB 的一些基本概念。

主机（host）：USB 系统中的所有的数据通信（不论是上行通信还是下行通信）都由 USB 主机启动，所以 USB 主机在整个数据传输过程中占据着主导地位。

设备（Device）：是 USB 系统中的数据终端，不能主动启动数据通信。它由一个或多个配置组成。由设备描述符说明设备的总体信息，并指明其所含的配置的个数。一个 USB 设备只能有一个设备描述符。

配置（Configuration）：一个 USB 设备可以包含一个或多个配置。在使用 USB 设备之前，必须为其选择一个合适的配置。配置描述符用于说明 USB 设备中各个配置的属性，如配置所含接口的个数等。USB 设备的每一个配置都必须有一个配置描述符。

接口（Interface）：一个配置可以包含一个或多个接口。接口是端点的集合。接口描述符用于说明 USB 设备中各个接口的特性，如接口所属的设备类及其子类。USB 设备的每一个接口都必须有一个接口描述符。

端点（Endpoint）：是 USB 设备中的实际物理单元。通常设备端点是内层的一块区域，或是控制器芯片内的一个寄存器，用来作为数据缓冲区。USB 数据传输就是在主机出发点和 USB 设备各个端点之间进行的。端点一般由 USB 接口芯片提供，USB 设备中的每一个端点都有唯一的端点号，每个端点所支持的数据传输方向一般而言也是确定的：输入（IN）或是输出（OUT）。也有些芯片提供的端点的数据方向是可以配置的。利用设备地址、端点号和传输方向就可以指定一个端点，并和它进行通信。0 号端点比较特殊，它有数据输入和数据输出两个物理单元，且只能支持控制传输。

字符串（String）：在 USB 设备中通常还含有字符串描述符，用以说明一些专用信息，如制造商的名称、设备的序列号等。它的内容以统一字符编码标准（UNICODE）的形式给出，且可以被用户软件读取。对 USB 设备来说，字符串描述符是可选的。

管道（Pipe）：在 USB 系统结构中，可以认为数据传输是在主机软件（USB 系统软件或用户软件）和 USB 设备的各个端点之间直接进行的。它们之间的连接称为管道。管道是在 USB 设备的配置过程中建立的。管道是对主机和 USB 设备间通信流的抽象。它表示主

机的数据缓存区和USB 设备的端点之间存在着逻辑数据传输，而实际的数据传输是由USB 接口层来完成的。

管道和 USB 设备中的端点一一对应。一个 USB 设备含有多少个端点，其和主机进行通信时就可以使用多少条管道，且端点的类型决定了管道中数据的传输类型。传输类型在后面会介绍。不论存在着多少条管道，在各条管道中进行的数据传输都是相互独立的。

**2. USB的信号表达机制与编码**

（1）USB 的信号表达机制

传统的传输方式大多采用“正信号”或“负信号”的二进制表达机制，这些信号利用单线传输。用不同的信号电平范围来表示 1 和 0，它们之间有一个阈值：如果在数据传输过程中遇到比较强的干扰，高低电平不会超过阈值，这样数据可以正常传输；如果在数据传输过程中遇到比较强的干扰，高低电平就有可能超过阈值，这样就会出现数据错误。一般情况是，总线的频率越高，线路之间的电磁干扰就越严重，数据失败的概率也越高，这样一来，传统信号表达机制就不再适合用于高速 USB 了。

USB 采用差分信号来传输数据。差分信号技术最大的特点是：必须使用两条线路才能表达一个比特位，用两条线路的压差作为判断 1 和 0 的依据。这样做的优点是具有极强的抗干扰能力。假如遇到外界强烈的干扰，两条线路对应的电平同样就会保持相对稳定，因此数据的准确性不会因干扰而有所降低。当然，由于一个比特位需要两条线路，所以在总线宽带相同的情况下，差分信号技术需要的时间是上述传输方式的两倍，因此这种方式适用于高速 USB。

（2）USB 的编码机制

USB 采用反向不归零编码方式（NRZI）。使用反向不归零方式能保证数据的完整性，并且不需要在传输过程中有独立的始终信号，图 22-2 给出了反向不归零编码方式的数据和编码之后的结果。在采用反向不归零编码方式时，遇到 0 转换，遇到 1 保持。反向不归零编码必须保持与输入数据的同步，以确保数据采样正确。反向不归零编码数据流必须在一个数据窗口被采样，无论前一个位时间是否发生过转换，编码器在每个位时间都会进行数据采样以检查是否有转换。

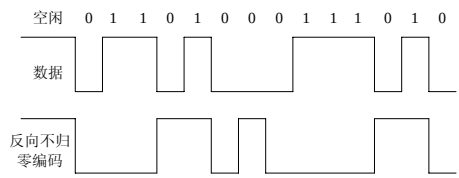



图 22-2　反向不归零编码方式的数据和编码

但是这样会出现一个很严重的问题，那就是若重复相同的 1 信号一直进入时，就会造成数据长时间无法转换，逐渐地积累而导致“塞车”现象，从而使得读取的时序发生错误。所以在反向不归零编码之间，还需要执行所谓的位填充操作。

（3）位填充

位填充要求如果数据流中连续出现 6 个 1 就要强行转换。这样，接收器在反向不归零编码数据流中最多每 7 个位就会检测到一次跳转。这样就保证了接收器与输入数据流保持同步。反向不归零编码的发送器要把 0（填充位）插到数据流中，如图 22-3 所示。所以接收器就必须在连续 6 个 1 之后识别出跳转并把此 0 位扔掉。

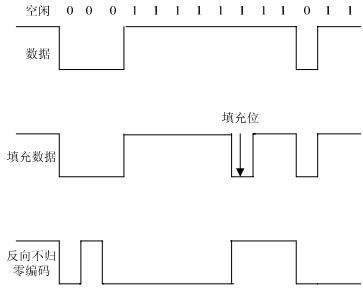



图 22-3　在 USB 电缆上使用反向不归零编码和差分信号的传输

## 22.3　硬件电路设计

本系统的硬件电路主要有电源电路、复位电路、仿真调试电路、USB 电路、加速度传感器电路、人机接口电路和单片机电路。下面主要介绍 USB 电路、加速度传感器电路、人机接口电路和单片机电路的设计。

### 22.3.1　USB 电路

USB 电路包含了 Host 和 Device 两种模式，USB 数据线的下拉控制通过 I/O 端口 PTB4 和 PTB5 来控制。ADM1869L 为 2 安培 P 通道开关，用来控制 USB 处于 Host 模式时给 Device 设备的供电控制。USB 电路的电路原理图如图 22-4 所示。

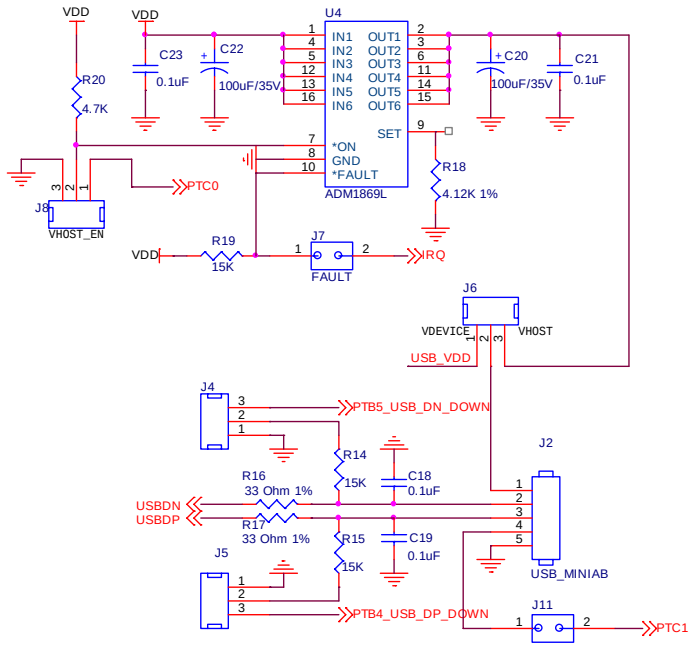



图 22-4　USB 电路原理图

## 22.3.2 加速度传感器电路

加速度传感器 MMA7260 用来测量 x、y、z 三个方向上的加速度值，可测加速度值为 1.5g、2g、4g 和 6g，其电路原理图如图 22-5 所示。

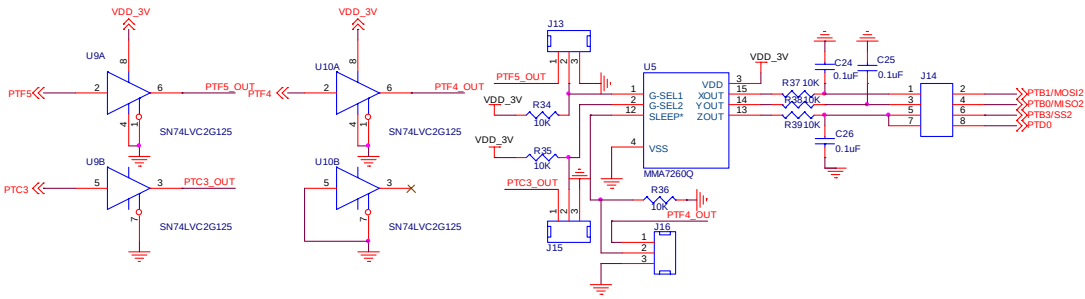



图 22-5 加速度传感器电路图

如图 22-6 所示，I/O 端口 PTF5 和 PTC3 控制两个片选，PTF4 控制 MMA7260 的休眠工作模式，MMA7260 通过 SPI 和单片机接口。

## 22.3.3 人机接口电路

人机接口主要包括按键和 LED，本例扩展了 4 个独立按键和 8 个 LED 发光二级。4 个按键连接在引脚 PTG0～PTG3。人机接口电路如图 22-6 所示。

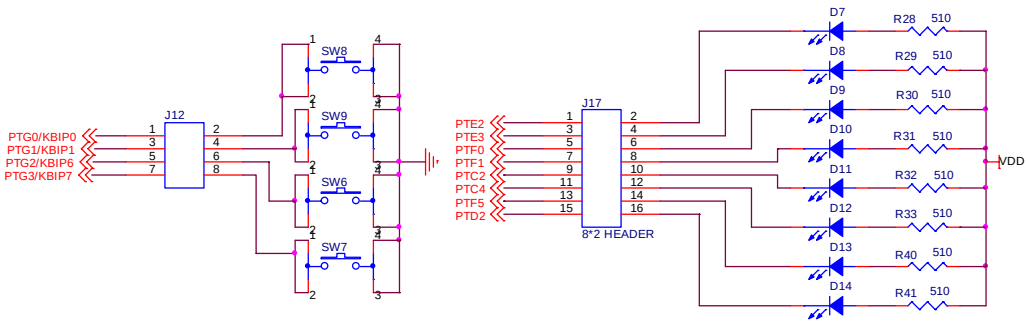



图 22-6 人机接口电路原理图

## 22.3.4 单片机系统

单片机选用 Freescale 公司的 MC9S08JM60，则需将 USB 电路、加速度传感器电路、人机接口电路分别与 MC9S08JM60 接口。如图 22-7 所示为具体的单片机系统电路图。

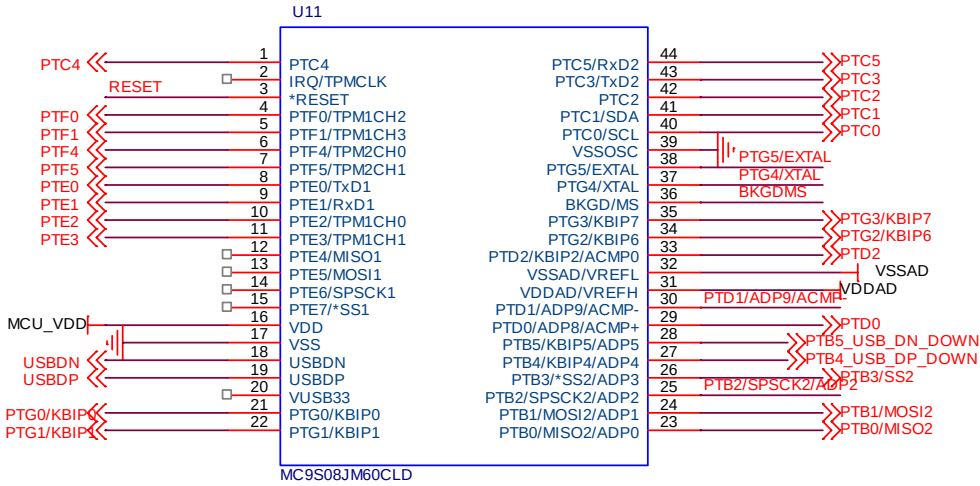



图 22-7　单片机系统电路图

由图 22-7 可以看出，单片机的电源电路、仿真调试电路等可参考第 5 章相关部分，在此不作介绍。考虑到电源的输入纹波对单片机的影响，在电源的管脚上增加一个 104 的滤波电容，以减少干扰。

## 22.4　软件设计与代码

### 22.4.1　软件设计流程

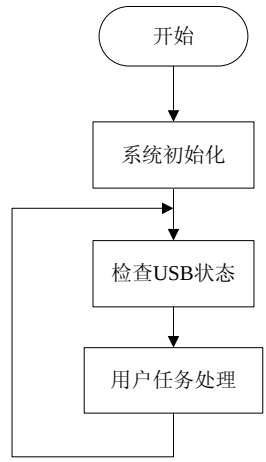



图 22-8　主程序流程图

USB 数据传输系统的软件设计主要包括以下几个部分：USB 协议栈、ADC、TPM 和 GPIO 驱动和应用层软件。USB 协议栈主要用于 USB 设备的枚举和配置，USB API 为访问 USB 设备提供程序接口，它基于 USB 协议栈，提供 USB 模块端点 1～6 的通信功能，应用层调用 API 来发送或接受数据。ADC、TPM 和 GPIO 驱动负责完成外设模块的初始化，为应用程序访问这些设备作好准备。USB 数据传输系统的主程序流程图如图 22-8 所示。

如图 22-8 所示，系统上电后，首先进行系统初始化，包括 MCG（多功能时钟发生器）、ADC、TPM、GPIO、USB 模块等外设的初始化，然后程序进入无限循环，检查 USB 状态函数控制 USB 设备的插入和挂起，用户任务处理函数负责处理 USB 设备枚举成功后事件的处理。

下面分别对每个部分进行具体的介绍。

### 22.4.2　USB 模块初始化程序

作为系统初始化中最重要的一步，USB 模块初始化主要负责使能调压器、PHY、上拉

电阻等，以及初始化 BDT。下面给出该部分的具体程序。

```
void  Initialize_USBModule()
{
  char i;
  unsigned char *pUsbMem = (unsigned char *) &Bdtmap;

  Usb_Device_State = POWERED_STATE;
  #ifdef SELF_POWER
    Usb_Stat._byte = 0x01;              //自供电
  #else
    Usb_Stat._byte = 0x00;              //总线供电
  #endif
  Usb_Active_Cfg = 0x00;

  USBCTL0 = USB_RST;
  while(USBCTL0_USBRESET)
  {
  };

  for(i= 0; i < 0xFF; i++)          //USB RAM 初始化，用于测试目的
  {
    *pUsbMem = 0x00;
    pUsbMem ++;
  }

  /* BDT 初始化部分*/
  Bdtmap.ep0Bo.Stat._byte = _SIE|_DATA0|_DTS;  //端点 0 输出初始化
  Bdtmap.ep0Bo.Cnt = EP0_BUFF_SIZE;
  Bdtmap.ep0Bo.Addr = 0x0C;
  Bdtmap.ep0Bi.Stat._byte = _CPU;              //端点 0 输入缓冲器初始化
  Bdtmap.ep0Bi.Cnt = 0x00;
  Bdtmap.ep0Bi.Addr = 0x08;

  USBCTL0 = 0X00;
  EPCTL0 = 0x0D;                               //使能端点 0
  INTENB = 0x01;
}
```

## 22.4.3 USB 模块的相关程序

**1. USB模块状态检测程序**

USB 模块状态检测主要用来检测是否有 USB 设备接入 USB 总线。具体的程序如下。

```
void Check_USB_Status(void)
{
   #ifdef SELF_POWER
```

```
    if(PTGD_PTGD0 == USB_BUS_ATTACHED)        //JM60 已经挂接在 USB 总线上?
    {
    #endif
        if(CTL_USBEN == 0)                    //JM60 模块关闭?
         {
           EPCTL0 = 0x0D;
           INTSTAT = 0xBF;
           CTL = 0x00;
           INTENB = 0x00;                    //禁止 USB 中断
           CTL_USBEN = 0x01;                  //使能 USB 模块
           USBCTL0 = UCFG_VAL;                //将 JM60 挂接在 USB 总线上

           Usb_Device_State = ATTACHED_STATE;
         }

   #ifdef SELF_POEWER
     }
    else
    {
        if(CTL_USBEN == 1)                    //JM60 USB 使能?
        {
            CTL = 0x00;                      //禁止模块
            INTENB = 0x00;                    //屏蔽 USB 中断
            Usb_Device_State = POWERED_STATE;
         }
    }
   #endif

     if(Usb_Device_State == ATTACHED_STATE)
      {
        INTSTAT = 0xBF;                      //清除 USB 中断标志位
        INTENB = 0xBF;
      }

    if(Usb_Device_State == USB_ENTER_SUSPEND)
    {
       USB_Suspend();
    }
}
```

**2. 用户任务处理程序**

用户处理程序根据接收到的命令，解析命令包，根据解析的结果执行相应的子程序，并将执行结果反馈给主机。下面给出用户任务处理程序具体的程序。

```
void Usr_Task(void)
{
```

```
if(Data_Logging)
{
  if(ADC_Results_Available)
  {
     if(Working_Buf)
     {
      if(Is_Usr_Ep_Available(5,0))
      {
        ADC_Results_Available = 0;
        Send_Data_PpEp(0x05,0,ADC_Cvt_Result_Buf0, Valid_Data_Len);
      }
     }
     else
     {
      if(Is_Usr_Ep_Available(5,1))
      {
        ADC_Results_Available = 0;
        Send_Data_PpEp(0x05,1,ADC_Cvt_Result_Buf1, Valid_Data_Len);
      }
     }

  }
}

if(Get_Ep_State(1,0))
{
  Usr_Buf2[0] = Usr_Buf1[0];
  Usr_Buf2[1] = Usr_Buf1[1];

  switch(Usr_Buf1[0])
  {
    case 0:                          //MCU
      if(Usr_Buf1[1] == 0x01)
      {
        while(1);                    //通过看门狗复位 MCU
      }
      else
      {
         Usr_Buf2[2] = 0x00;
         Usr_Buf_Len[1] = 3;
      }
    break;

    case 1:                           //Boot Loader
      if(Usr_Buf1[1] == 0x01)
      {
```

```
    while(1);                    //通过看门狗复位 MCU

  }
  else
  {
    Usr_Buf2[2] = 0x00;
    Usr_Buf_Len[1] = 3;
  }
break;

case 2:                         //ADC
  switch(Usr_Buf1[1])
  {
    case 0x00:                  //获取 ADC 转换的结果
      Usr_Buf2[2] = Usr_Buf1[2];
      if(Get_ADC_Ch_Result(Usr_Buf2[2], &Usr_Buf2[3]))
      {
        Usr_Buf_Len[1] = 0x05;
      }

    break;

    case 0x01:                  //配置 ADC
      if(ADC_Config(Usr_Buf1[2], Usr_Buf1[3],Usr_Buf1[4],Usr_Buf1[5]))
      {
        Usr_Buf2[2] = 0xFF;
      }
      else
      {
        Usr_Buf2[2] = 0x00;
      }

      Set_Trigger_Type(Usr_Buf1[6], Usr_Buf1[7], Usr_Buf1[8],
                       Usr_Buf1[9]);
      Usr_Buf_Len[1] = 0x03;

    break;

    case 0x02:                  //返回 ADC 配置数据
      Get_ADC_Configuration(&Usr_Buf2[2]);
      Usr_Buf_Len[1] = 10;

    break;

    case 0x03:                  //启动 ADC
      if(ADC_Start())
      {
```

```
        Usr_Buf2[2] = 0xFF;
        Data_Logging = 1;
      }
      else
      {
        Usr_Buf2[2] = 0x00;
      }

      Usr_Buf_Len[1] = 0x03;

    break;

    case 0x04:                  //停止 ADC 转换
     ADC_Stop();
     Data_Logging = 0;

     Usr_Buf2[2] = 0xFF;
     Usr_Buf_Len[1] = 0x03;
    break;

    case 0x05:                  //返回 ADC 状态
      Usr_Buf2[2] = Get_Adc_State();
      Usr_Buf_Len[1] = 0x03;
    break;

    default:
      Usr_Buf2[2] = 0x00;
      Usr_Buf_Len[1] = 0x03;
    break;
  }
break;

case 6:                         //GPIO
   if(Usr_Buf1[1] == 0x00)
   {
     Led_OnOff(Usr_Buf1[2]);
     Usr_Buf2[2] = 0xFF;
     Usr_Buf_Len[1] = 3;
   }
   else
   if(Usr_Buf1[1] == 0x01)
   {
      Get_Button_Status(&Usr_Buf2[2]);
      Usr_Buf_Len[1] = 3;
   }
break;
```

```
    case 7:                          //定时器 TPM
       switch(Usr_Buf1[1])
       {
        case 0:                      //启动 PWM
           if(TPM_Config((int)((Usr_Buf1[2]<<8)| Usr_Buf1[3]),
                                Usr_Buf1[4])  )
           {
             TPM_Start();
             Usr_Buf2[2] = 0xFF;
             Usr_Buf_Len[1] = 3;
           }
           else
           {
             Usr_Buf2[2] = 0x00;
             Usr_Buf_Len[1] = 3;
           }


        break;

        case 1:                      //停止 PWM
           TPM_Stop();
           Usr_Buf2[2] = 0xFF;
           Usr_Buf_Len[1] = 3;
        break;

        default:
           Usr_Buf2[2] = 0x00;
           Usr_Buf_Len[1] = 3;
        break;
       }

    break;
  }

  Cmd_Exe_Flag = 1;
  Clr_Ep_State(1,0);    //清除标志位
}

/* 向主机传输数据 */
if(Cmd_Exe_Flag)
 {
   if(Is_Usr_Ep_Available(2,0))
   {
     Send_Data_Ep(2,Usr_Buf2,Usr_Buf_Len[1]);
     Set_Ep_State(2,0);
     Cmd_Exe_Flag = 0;
```

```
    }
   }

  return;
}
```

### 22.4.4 主程序

系统经过一系列初始化后，即进入主程序工作。主程序调用前面介绍的程序，完成 USB 设备的加载/卸载处理、用户任务处理等功能。在具体应用时，读者可以根据自己系统的需要来设计相应的程序。下面给出 USB 串行总线数据传输系统的主程序。

```
void main(void)
{

  Init_Sys();                          //初始化系统外设

  EnableInterrupts;                    //使能中断


  for(;;)
  {
    __RESET_WATCHDOG();                //喂狗

    Check_USB_Status();                //检查 USB 设备状态

    if(Usb_Device_State == CONFIGURED_STATE)
    {
      Usr_Task();                      //用户任务处理
    }
  }
 }
```

## 22.5 实例总结

本章详细介绍了基于 MC9S08JM60 单片机的 USB 模块实现的数据传输系统设计。MC9S08JM60 集成了 USB2.0 控制器，Freescale 公司提供了基于该单片机的 USB 协议栈，为用户设计基于该单片机的 USB 应用提供了捷径。本例利用该 USB 协议栈实现了一个简单的数据传输系统。虽然系统比较简单，但包括了 USB 程序设计的各个环节，如枚举、配置、数据传输等，代表性强，读者可以在本章内容的基础上，根据自己系统的需要，实现更加复杂的系统，如可以在此基础上实现 FAT 文件传输系统，从而实现 U 盘的功能等。

# 第三篇

# 综合系统设计实例篇

第 23 章　出租车计价器系统设计实例

第 24 章　手持式超声波测距仪系统设计实例

# 第 23 章

# 出租车计价器系统设计实例

出租车计价器是出租车中最重要的工具。随着电子技术的发展，计价器的功能也越来越强大，从刚开始的只显示路程（需要司机自己定价，计算后四舍五入），到能够自主计费，以及现在的能够打印发票和语音提示，按时间自主变动单价等功能。本章将介绍采用 MC9S08JM60 单片机实现的出租车计价器，用较少的硬件和适当的软件相互配合可以很容易地实现设计要求，且灵活性强，可以通过软件编程来完成更多的附加功能。

## 23.1 实例内容说明

出租车计价器的营运金额是营运里程与价格的函数（等候时间一般折算成一定比例的里程来计算）。出租车计价器通过传感器与行驶车辆连接。出租汽车的实际里程通过传感器的脉冲信号在计价器里折算成一定的计价营运里程。目前市场上的出租车计价器功能主要具有数据的复位功能、白天/晚上转换功能、数据输出功能、计时计价功能等，但能够进行语音播报数据信息的出租车计价器还是比较少见的。针对这一点我们来设计一款多功能出租车计价器，在原有功能的基础上增加单价输出、单价调整、路程输出、显示当前的系统时间、语音播报数据信息等功能。如图 23-1 所示为基于 MC9S08JM60 实现的出租车计价器的原理框图。

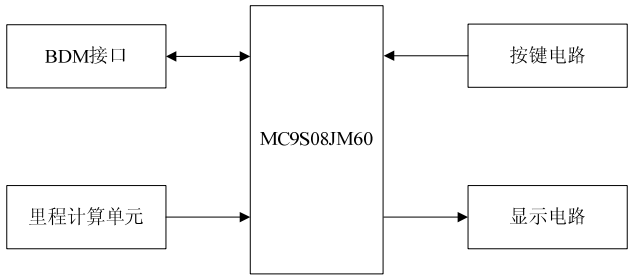



图 23-1　基于 MC9S08JM60 实现的出租车计价器原理框图

由图 23-1 可以看出，基于 JM60 实现的出租车计价器主要由 MC9S08JM60 单片机、里程计算单元、按键电路和显示电路等四部分组成。车轮每转一圈，里程传感器就检测并输出信号；里程计算单元将此信号处理后送入单片机，供后续计算；按键电路负责显示切换、功能设定等功能；显示电路负责显示单价、路程、总金额等相关信息。MC9S08JM60 单片机作为整个系统的控制器，实现出租车计价器的功能。

## 23.2　设计思路分析

基于 MC9S08JM60 实现的出租车计价器的控制器选用 Freescale 公司基于 HCS08 内核系列的单片机 MC9S08JM60。MC9S08JM60 内核频率为 48MHz，具有高达 60KB 片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、$I^2C$ 控制器、定时器等多种外设，方便进行系统设计。

按照图 23-1 所确定的系统原理框图，里程传感器是整个系统的关键，因此首先需选择合适的里程传感器。

里程传感器我们采用的是开关型霍尔传感器 A44E，其工作电压范围比较宽（4.5V~18V），其输出的信号符合 TTL 电平标准，可以直接接到单片机的 IO 端口上，而且其最高检测频率可达到 1MHz。

集成开关型霍尔传感器原理如图 23-2 所示。

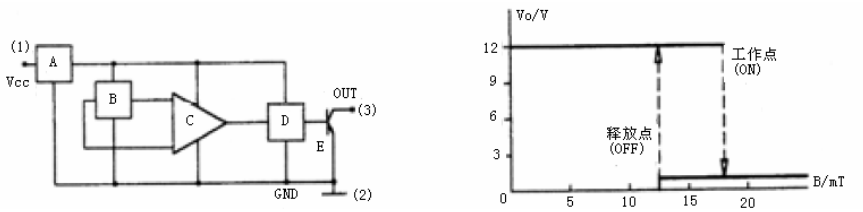



图 23-2　集成开关型霍尔传感器原理图

A44E 集成霍尔开关由稳压器 A、霍尔电势发生器（即硅霍尔片）B、差分放大器 C、施密特触发器 D 和 OC 门输出 E 五个基本部分组成。

在输入端输入电压 $V_{CC}$，经稳压器稳压后加在霍尔电势发生器的两端，根据霍尔效应原理，当霍尔片处在磁场中时，在垂直于磁场的方向通以电流，则与这二者相垂直的方向上将会产生霍尔电势差 $V_H$ 输出，该 $V_H$ 信号经放大器放大后送至施密特触发器整形，使其成为方波输送到 OC 门输出。当施加的磁场达到工作点（即 $B_{OP}$）时，触发器输出高电压（相对于低电位），使三极管导通，此时 OC 门输出端输出低电压，通常称这种状态为开。当施加的磁场达到释放点（即 $B_{rP}$）时，触发器输出低电压，三极管截止，使 OC 门输出高电压，这种状态为关。这样两次电压变换，使霍尔开关完成了一次开关动作。

# 23.3 硬件电路设计

出租车计价器系统主要有电源电路、复位电路、仿真调试电路、里程计算单元、显示电路、按键电路和单片机电路，下面进行具体介绍。

## 23.3.1 里程计算单元

里程计算单元将里程传感器的输出信号处理后送入单片机的 IO 口，其电路原理图如图 23-3 所示。

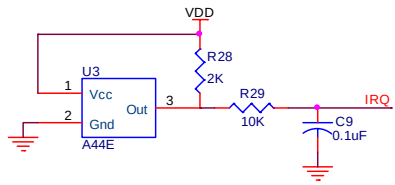



图 23-3 里程计算单元电路

如图 23-3 所示，里程计算单元的输出 IRQ 接 MC9S08JM60 的 PTG0 引脚，PTG0 设置成低电平中断，这样霍尔开关检测并输出信号，引起单片机的中断，对脉冲计数。为防止干扰触发 IRQ 中断请求，里程计算单元集成了一阶无源低通滤波器，其截止频率为 106Hz。车轮每转一圈（我们假设车轮的周长是 1 米），此滤波器可确保计价器适用于车速最高为 380km/h 的出租车上。

## 23.3.2 显示电路

计价器需有单价、路程、总金额显示输出。单片机系统中常用的显示输出有 LED 数码管和 LCD 液晶屏。采用 LCD 液晶段码显示时，在距离屏幕 1 米之外就无法看清数据，不能满足要求，而且在白天其对比度也不能满足要求，因此我们采用 6 位 LED 数码管的分屏显示，其电路原理如图 23-4 所示。

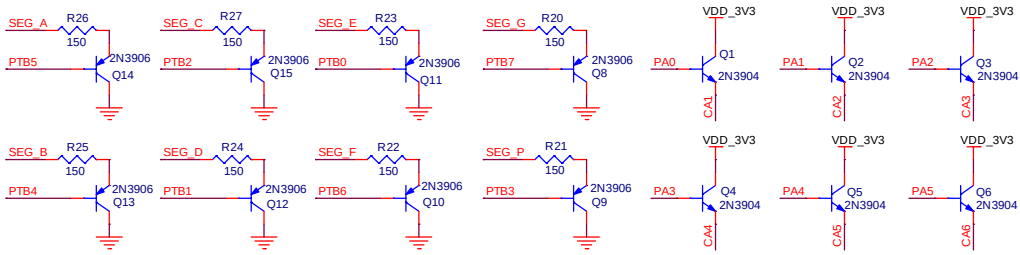



图 23-4　显示电路原理图

由图 23-4 可知，6 位 LED 数码管的位选线 CA1～CA6 接 MC9S08JM60 的 PA0～PA5 引脚，段选线 SEG_A～SEG_P 接 MC9S08JM60 的 PTB 口。由于 MC9S08JM60 的 IO 口驱动能力有限，所以所有的位选和段选信号都经三极管放大。

## 23.3.3　按键电路

本设计共采用 5 个独立按键 SW1、SW2、SW3、SW4、SW5，它们分别接 MC9S08JM60 的 PA0～PA4 引脚，按键主要用来处理计价器不同功能间的切换。其电路原理如图 23-5 所示。

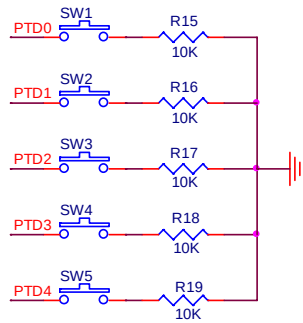



图 23-5　按键电路原理图

## 23.3.4　单片机系统

单片机选用 Freescale 公司的 MC9S08JM60，只需将单片机的 PTA 口、PTB 口、IRQ 引脚与按键电路、显示电路和里程计算单元电路接口即可。另外单片机还需要相应的振荡电路才能正常工作。如图 23-6 所示为具体的单片机系统电路图。

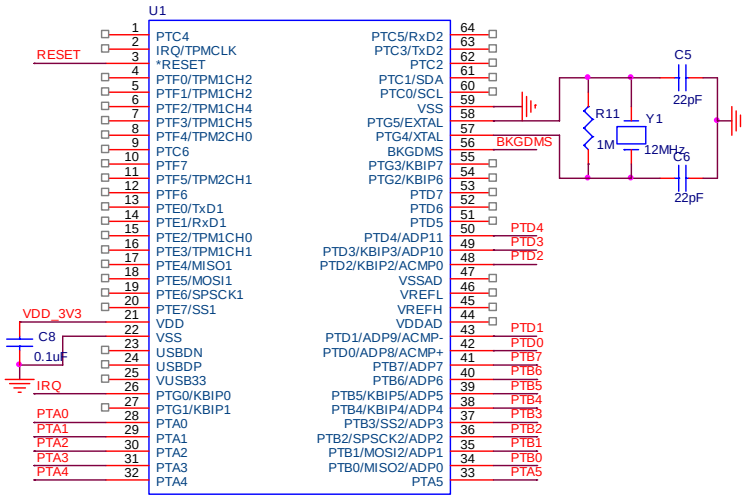



图 23-6　单片机系统电路图

由图 23-6 可以看出，该单片机电路非常简单。考虑到电源的输入纹波对单片机的影响，在电源的管脚上增加一个 104 的滤波电容，以减少干扰。

# 23.4 软件设计与代码

## 23.4.1 软件设计流程

基于 MC9S08JM60 的出租车计价器系统软件分三层设计。第三层即应用层，基于状态机设计，事件驱动方式实现具体的功能，其状态流程图如图 23-7 所示；第二层为应用程序接口层，负责和硬件接口，处理定时、中断和管理硬件（显示、按键）等，并为上层提供服务；第一层，即硬件接口层。采用分层结构，系统软件结构清晰、移植性强。

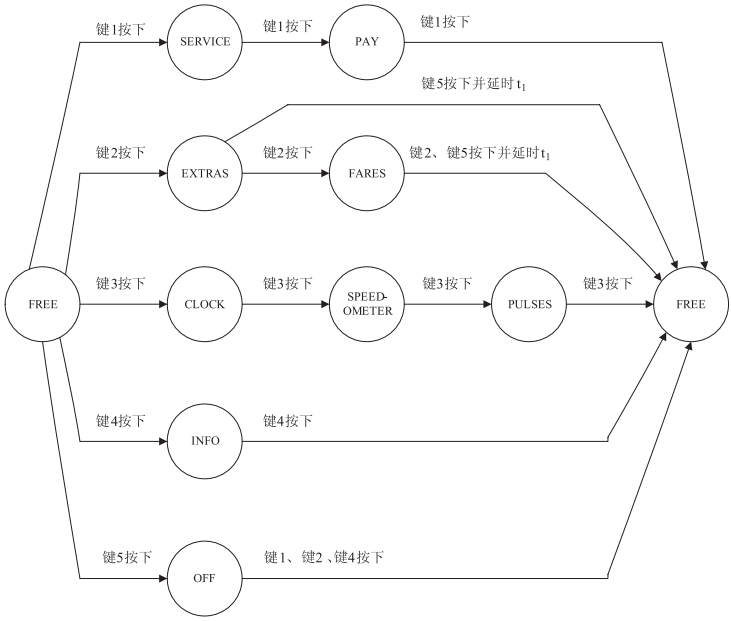



图 23-7　应用层软件流程图

下面分别对每个层进行具体的介绍。

## 23.4.2 硬件层程序

硬件层负责初始化硬件，主要包括端口初始化、定时器初始化、外部中断初始化和系统变量初始化。所有初始化完成后，上层应用程序即可使用这些资源。硬件层初始化程序主要包括两个子程序：void PE_level_init(void)和 void System_Initialize (void)。其具体程序如下：

```
void PE_low_level_init(void)
{
  /* CPU 寄存器的初始化，主要是 IO 寄存器 */
```

```
    clrReg8Bits(PTGDD, 0x01);
    setReg8Bits(PTGPE, 0x01);
    clrReg8Bits(PTASE, 0x3F);
    setReg8(PTBSE, 0x00);
    clrReg8Bits(PTCSE, 0x7F);
    setReg8(PTDSE, 0x00);
    setReg8(PTESE, 0x00);
    setReg8(PTFSE, 0x00);
    clrReg8Bits(PTGSE, 0x3F);
    setReg8(PTADS, 0x3F);
    setReg8(PTBDS, 0xFF);
    setReg8(PTCDS, 0x7F);
    setReg8(PTDDS, 0xFF);
    setReg8(PTEDS, 0xFF);
    setReg8(PTFDS, 0xFF);
    setReg8(PTGDS, 0x3F);
    Timer_1ms_Init();            //定时器初始化

   /* 外部中断初始化*/
   KBIPE |= 0x01;
   setReg8(KBISC, 0x00);
   KBISC_KBACK = 0x01;          // 清除中断标志
   KBISC_KBIE = 0x01;
  }

void System_Initialize (void)
{
    current_state = FREE;       //起始状态为 Free 状态
    /* 显示相关变量初始化*/
    PTA = 0x00;                 // 显示使能
    DDRA = 0xFF;                // 端口 A 为输出
    PTB = DISPLAY_OFF;          // 关显示
    DDRB = 0xFF;                // 端口 B 为输出

    read_buttons_time = 0;      // 用于读按键操作的时间变量
    arr_display_index = 0;      // 选择显示 0
    display_en = 0x01;          // 使能显示 0
    pushed_buttons = NO_BUTTON; // 按下的按键

    /* 费率相关变量初始 */
    fare_type = DAY;                    // 默认费率为 DAY
    fare_active = FARE_TABLE[FARE_DAY]; // 有效费率由 FARE_TABLE 决定
    arr_display[0] = TRANSLATE_NUMBER[DAY+1];
    ini_ch_active = INI_CH_TABLE[INI_CH_DAY];// 初始收费
```

```
    /* 计数器和累加器变量初始化 */
    counter_ms = 0;                      // 计数器复位
    counter_pulses = 0;
    accumulator_pulses = 0;              // 累加器复位
    accumulator_time = 0;
    accumulator_trip = 0;
    accumulator_inc = 0;
    accumulator_distance_traveled = 0;
    accumulator_extras = 0;
    clock_ms = 0;
    clock_minutes = 0;
    clock_hours = 12;            // 12: 00 启动

/* Incremental-charge related variables */
    flag_charge = NO_CHARGE;
    inc_time = 5000;
    inc_distance = 45;
    return;
}
```

### 23.4.3 应用程序接口层程序

应用程序接口层为上层——应用层提供服务，主要包括变量清零子程序 CleanUp、显示子程序等。下面给出应用程序接口层的具体程序。

```
void CleanUp(void)
{
    accumulator_extras = 0;
    accumulator_trip = 0;
    accumulator_pulses = 0;
    accumulator_time = 0;
    accumulator_distance_traveled = 0;
    accumulator_inc = 0;
    return;
}

void DisplayMsg(const Byte *_str)
{
    Byte _i;
    Byte _temp;
    Byte _selector_display;

    for(_i = 0, _selector_display = 1 ; _i<5 ; _i++, _selector_display++)
```

```
 {
       _temp = *(_str + _i);
       arr_display[_selector_display]=TRANSLATE_LETTER[(_temp - 32)];
   }
   return;
}

void DisplayNum(ulong _num)
{
   Byte _i;
   Word _short_num;
   ulong _temp;

   for (_i = 5; _i > 0; _i--) {        // 显示 5 次
          _temp = (ulong)(_num / 10);
          _short_num = _temp;         // 备份系数
          _temp *= 10;                // 获取除最后一位的数
          _temp = _num - _temp;       // 获取后几位
          _num = _short_num;
          arr_display[_i] = TRANSLATE_NUMBER[(Byte)_temp];
          if (_i == 3)
{
              arr_display[_i] &= SEG_P;// turn on P segment
          }
   }
   return;
}
```

### 23.4.4　应用层主程序

应用层基于状态机设计，状态机共有 11 个状态，每个状态的具体功能见表 23-1。

表 23-1　状态机状态功能表

| 状　态 | 描　述 |
| --- | --- |
| FREE | 空闲状态 |
| IN_SERVICE | 禁止其他功能<br>备份未服务时的里程数据<br>显示已选择的费率类型<br>显示费用（费用在每个固定的时间间隔或每个固定的里程后增加） |
| PAY | 备份服务时的里程数据，总收入<br>循环显示符号“PAY”和费用 |
| EXTRAS | 在指定的时间显示符号“EXTRA”<br>可选择外部收费类型 |

续表

| 状　态 | 描　述 |
|---|---|
| FARES | 在指定的时间显示符号“FARE”<br>允许选择费率类型 |
| CLOCK | 显示时间 |
| SPEEDOMETER | 显示速度 |
| PULSES | 记录车轮已转圈数 |
| INFO | 显示计价器的重要信息和累加器的信息<br>累加器信息显示按如下顺序：<br>已行驶的总里程<br>服务时行驶的总里程<br>总增加的费用<br>总收入 |
| PROGRAM | 在指定时间显示符号“PROGR” |
| OFF | 在指定时间显示符号“OFF”<br>关闭显示 |

状态间的切换由 5 个按键触发，表 23-2 显示了在不同状态时按键触发的动作。

表 23-2　按键动作转换表

| 状　态 | 按键 1 | 按键 2 | 按键 3 | 按键 4 | 按键 5 |
|---|---|---|---|---|---|
| FREE | SERVICE | FARES | CLOCK | INFO | PROGRAM \| OFF |
| SERVICE | PAY | - | - | - | - |
| PAY | FARES | - | - | - | - |
| EXTRAS | - | FARES | extra++ | - | Select extra;FREE |
| FARES | - | FREE | fare++ | - | Select fare;FREE |
| CLOCK | hours++ | minutes++ | SPEEDO-METER | - | - |
| SPEEDOMETER | - | - | PULSES | - | - |
| PULSES | reset | freeze | FREE | - | - |
| INFO | - | - | info++ | FREE | - |
| PROGRAM | - | - | - | - | - |
| OFF | FREE | FREE | FREE | FREE | - |

应用层主函数和子程序如下所示。

```
void main (void){
   PE_low_level_init();              //外设初始化
   System_Initialize();              //系统变量初始化
   for(;;)
   {
      switch(current_state)
```

```
        {
               case FREE:       Free();
                                break;
               case SERVICE:    Service();
                                break;
               case PAY:        Pay();
                                break;
               case EXTRAS:     Extras();
                                break;
               case FARES:      Fares();
                                break;
               case CLOCK:      Clock();
                                break;
               case SPEEDOMETER: Speedometer();
                                break;
               case PULSES:     Pulses();
                                break;
               case INFO:       Info();
                                break;
              case OFF:        Off();
                                break;
               default:         current_state = FREE;
                                break;
        }
    }
 }

void Free(void)
{
    WaitForButtonsRelease();
    DisplayMsg(FREE_STATE_MSG);           // 显示"FREE"
    if (FLASH_PROGRAM != UNCORRUPT_FLASH) //FLASH 是否崩溃?
    {
        DisplayMsg(ERROR_MSG);
        while(pushed_buttons != BUTTON5){;}
    }

    while(1)                              //等待按键按下
   {
        switch(pushed_buttons) {         // 根据按键切换状态
           case BUTTON1:    current_state = SERVICE;
                        return;
           case BUTTON2:    current_state = EXTRAS;
                        return;
```

```
            case BUTTON3:   current_state = CLOCK;
                            return;
            case BUTTON4:   current_state = INFO;
                            return;
            case BUTTON5:   current_state = PROGRAM;
                            return;
        }
    }
}

void Service(void)
{
    char *_ptrMsg;

    CleanUp();
    _ptrMsg = &FARE_MESSAGE_TABLE[(fare_type * 5)];
    DisplayMsg(_ptrMsg);            // 显示每秒的费率
    WaitForButtonsRelease();
    Delay(1000, BUTTON1);           // 等待一秒
    accumulator_trip = accumulator_extras + ini_ch_active;
    WaitForButtonsRelease();
    flag_charge = CHARGE;           // 开始计费
    while (pushed_buttons != BUTTON1)//一直刷新将付的数额，直到用户按下按键 1
   {
            DisplayNum(accumulator_trip);
    }
    current_state = PAY;            // 返回 PAY 状态
    DisplayMsg(PAY_STATE_MSG);       // 显示符号 "PAY"
    return;
}

void Pay(void)
{

    flag_charge = NO_CHARGE;         // 停止计费
    WaitForButtonsRelease();
    while (pushed_buttons != BUTTON1) //循环显示符号“PAY”和要付的总额，
                                      //直到按下按键 1
   {
            DisplayMsg(PAY_STATE_MSG);
            Delay(2000, BUTTON1);
            DisplayNum(accumulator_trip);
            Delay(2000, BUTTON1);
    }
```

```
    current_state = FREE;              // 返回 FREE 状态
    CleanUp();                         // 重新初始化累加器
    return;
}

void Extras(void)
{
    Byte _extra_ch_temp = 1;         // 额外的计费范围从 1 到 9

    DisplayMsg(EXTRA_STATE_MSG1);     // 显示“PLUS”
    WaitForButtonsRelease();
    Delay(1000, BUTTON2&BUTTON3);
    DisplayMsg(EXTRA_STATE_MSG2);      // 默认状态，额外计费起步为#1
    WaitForButtonsRelease();
    while (-1)
    {
        /*如果 10s 内，用户未有操作，则返回 FREE 状态*/
        Delay(10000, BUTTON2&BUTTON3&BUTTON5);
        if (pushed_buttons != NO_BUTTON)  //是否有键按下?
       {
            if (pushed_buttons == BUTTON2) //按键 2 按下，则返回 FARES 状态
            {
                current_state = FARES;
                return;
            }
            if(pushed_buttons == BUTTON5)//按键 5 按下，则处理
                                         //accumulator_extras
            {
                if (_extra_ch_temp == 0)
                {
                    accumulator_extras = 0;  // 擦除 accumulator_extras 值
                } else
                {
                    accumulator_extras += EXTRA_TABLE[_extra_ch_temp-1];
                }
                current_state = FREE;
                return;
            }
            if (pushed_buttons == BUTTON3) //按键 3 按下，则返回 extra 计费
           {
                _extra_ch_temp++;
                if (_extra_ch_temp > EXTRAS_NUMBER)
                {
                    _extra_ch_temp = 0;
```

```
                }
                if (_extra_ch_temp == 0)
                {
                    DisplayMsg(EXTRA_STATE_MSG_ERASE);
                }
                else
                {
                    DisplayMsg(EXTRA_STATE_MSG2);
                    arr_display[5] = TRANSLATE_NUMBER[_extra_ch_temp];
                }
                WaitForButtonsRelease();
            }
        }
        else                          //过去 10s 内无键按下
        {
            current_state = FREE;     // 返回 FREE 状态
            return;
        }
    }
}

void Fares(void)
{
    Byte *_tarMsg;

    _tarMsg = &FARE_MESSAGE_TABLE[(fare_type * 5)];
    DisplayMsg(FARE_STATE_MSG);        // 显示符号 "FARE " 1s
    WaitForButtonsRelease();
    Delay(500, BUTTON2&BUTTON3);       // 延时 1s
    DisplayMsg(_tarMsg);               // 显示 active-fare 信息
    WaitForButtonsRelease();
    while (-1)
   {
      /*如果 10s 内，用户未有操作，则返回 FREE 状态*/
        Delay(10000, BUTTON2&BUTTON3&BUTTON5);
        if (pushed_buttons != NO_BUTTON) //如果有键按下
        {
            if (pushed_buttons == BUTTON2) //按键 2 按下，则返回 FREE 状态
            {
                current_state = FREE;
                return;
            }
            if (pushed_buttons == BUTTON5) //按键 5 按下，则返回 active fare
            {
```

```
                fare_active = FARE_TABLE[fare_type];
                ini_ch_active = INI_CH_TABLE[fare_type];
                arr_display[0] = TRANSLATE_NUMBER[fare_type+1];
                current_state = FREE;
                return;
            }
            if (pushed_buttons == BUTTON3) //按键 3 按下，则返回 next fare
            {
                fare_type++;
                if (fare_type >= FARES_NUMBER)
                {
                    fare_type = 0;
                }
                _tarMsg = &FARE_MESSAGE_TABLE[(fare_type * 5)];
                DisplayMsg(_tarMsg);
                WaitForButtonsRelease();
            }
        }
        else                              // 10s 内无键按下
        {
            current_state = FREE;         // 返回 FREE 状态
            return;
        }
    }
}

void Clock(void)
{
    unsigned long _time;

    DisplayMsg(CLOCK_STATE_MSG);       // 显示符号"TIME " 1s
    WaitForButtonsRelease();
    Delay(1000, BUTTON3);
    while(pushed_buttons != BUTTON3) //显示时间，直到按键 3 被按下
   {
        _time = (clock_hours * 100) + clock_minutes;
        DisplayNum(_time);
        if (pushed_buttons == BUTTON1) //按键 1 被按下，则 Hours ++
        {
            clock_hours++;
            if (clock_hours >= 24)
            {
                clock_hours = 0;
            }
```

```
            WaitForButtonsRelease();
        }
        if (pushed_buttons == BUTTON2) //按键 2 被按下，则 Minutes ++
        {
            clock_minutes++;
            if (clock_minutes >= 60)
            {
                clock_minutes = 0;
            }
            WaitForButtonsRelease();
        }
    }
    current_state = SPEEDOMETER;        //返回 SPEEDOMETER 状态
}

void Speedometer(void)
{
    Byte _current = 0;                   // 当前时间间隔内的脉冲数
    Byte _previous = 0;                  // 前个时间间隔内的脉冲数
    unsigned long _spd;                 // velocidad = 0
    const int _time = 1500;             // 每次采样的时间
    const int _factor = 240;            // 转换成 km/h

    DisplayMsg(SPEED_STATE_MSG);        // 显示符号"SPEED" 1s
    WaitForButtonsRelease();
    Delay(1000, BUTTON3);
    counter_ms = 0;                      // 重设通用计数器
    counter_pulses = 0;                 // 用于距离和时间
    while(pushed_buttons != BUTTON3) //显示速度，直至按键 3 被按下
   {
        _spd = (unsigned long) ((_current + _previous) / 2) * _factor;
        DisplayNum(_spd);
        if(counter_ms >= _time)
       {
            _previous = _current;
            _current = (Byte) counter_pulses;
            counter_pulses = 0;
            counter_ms = 0;
        }
    }
    current_state = PULSES;              // 返回 PULSES 状态
}

void Pulses(void)
```

```
{
    DisplayMsg(PULSE_STATE_MSG);          // 显示符号 "PULSE" 1 s
    WaitForButtonsRelease();
    Delay(1000,BUTTON3);                  // 延时 1s
    WaitForButtonsRelease();
    counter_pulses = 0;
    while (pushed_buttons != BUTTON3) //显示接收到的脉冲数，直到按键 3 被按下
    {
        DisplayNum(counter_pulses);
        if (pushed_buttons == BUTTON1) //按键 1 被按下，则复位
        {
            counter_pulses = 0;
        }
        if (pushed_buttons == BUTTON2) //按键 2 被按下，则 freeze
        {
            Delay(20000,BUTTON1);
        }
    }
    current_state = FREE;                 // 返回 FREE 状态
}

void Info(void)
{
    Byte _char_offset=0;
    Byte _message_offset=0;
    ulong *_acc_table;

    _acc_table = &TOTAL_DISTANCE_TRAVELED;
    DisplayMsg(INFO_STATE_MSG);           // 显示符号 "INFOR" 1 s
    WaitForButtonsRelease();
    Delay(1000,BUTTON4);                  // 延时 1 s
    WaitForButtonsRelease();
    while (pushed_buttons != BUTTON4) //显示信息，直至按键 4 被按下
    {
        if(_message_offset < 24)
        {
            DisplayMsg(&INFO_TABLE[_message_offset + _char_offset]);
            _char_offset++;
            if (_char_offset > 3) // 因为消息是 8 字节，所以需滚动 3 倍
            {
                _char_offset = 0;
            }
        }
        else
```

```
        {
            DisplayNum(*_acc_table + 1);
        }
        Delay (800, BUTTON4&BUTTON3);
        if (pushed_buttons == BUTTON3) //按键 3 被按下，显示下一条消息
        {
            WaitForButtonsRelease();
            _message_offset += LENGTH_MESSAGE;
            _char_offset = 0;
            if (_message_offset > LENGTH_3_MESSAGES)
            {
                _acc_table++;              // 指向下一个累加器
            }
            if (_message_offset == LENGTH_8_MESSAGES) //超过 7 条消息
            {
                _message_offset = 0;
                _char_offset = 0;
                _acc_table = &TOTAL_DISTANCE_TRAVELED;
            }
        }
    }
    current_state = FREE;
}

void Off(void)
{
    DisplayMsg(EMPTY_MSG);
    arr_display[0] = 0xFF;
    WaitForButtonsRelease();
    while ((pushed_buttons == NO_BUTTON) || pushed_buttons == BUTTON5)
    {;}
    arr_display[0] = TRANSLATE_NUMBER[fare_type+1];
    current_state = FREE;
}
```

## 23.5 实例总结

本章详细介绍了基于 MC9S08JM60 单片机的出租车里程计的设计。充分利用 JM60 丰富的外设资源，从而简化了硬件设计；软件设计采用分层设计的思想，实现了软件的模块化和标准化。这个设计可移植性非常高。读者学习完后可以根据自己的需要实现更加复杂的出租车里程计系统，如增加语音提示、票据打印等功能。

# 第 24 章

# 手持式超声波测距仪系统设计实例

随着社会的发展，人们对距离或长度测量的要求越来越高。由于超声波测距是一种非接触检测技术，不受光线、被测对象颜色等的影响，在较恶劣的环境（如含粉尘）中具有一定的适应能力，因此越来越被人们所重视，应用范围也越来越广。本章将介绍采用 MC9S08JM60 单片机实现的手持式超声波测距仪系统， 并详细介绍系统的软硬件设计。

## 24.1 实例内容说明

超声波测距仪是根据超声波遇到障碍物反射回来的特性进行测量的。超声波发射器向某一方向发射超声波，遇到障碍物则反射回波，从而测出发射和接收回波的时间差 $t$，然后求出距离 $S=C\times t/2$，式中的 $C$ 为超声波波速。由于超声波也是一种声波，其声速 $C$ 与温度有关。在使用时，如果温度变化不大，则可认为声速是基本不变的。如果测距精度要求较高，则应通过温度补偿的方法加以校正。声速确定后，只要测得超声波往返的时间，即可求得距离。如图 24-1 所示为基于 MC9S08JM60 实现的手持式超声波测距仪的原理框图。

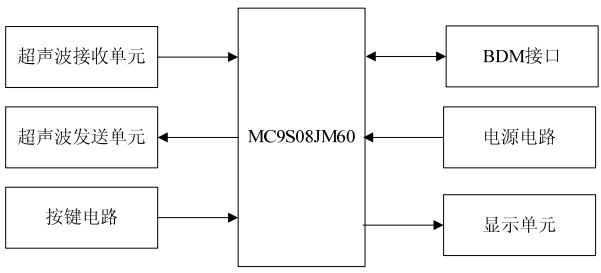



图 24-1　手持式超声波测距仪原理框图

由图24-1可以看出，基于MC9S08JM60实现的手持式超声波测距仪主要由MC9S08JM60单片机部分、发射单元、接收单元、按键电路、显示单元五部分组成。发射单元负责产生所需的超声波，并放大超声波，以确保传输尽可能远的距离。接收单元负责处理接收到的返回波，并将接收到的波转换成单片机能处理的信号。显示单元负责显示工作模式、测量的结果等。按键电路负责工作模式的切换。

## 24.2 设计思路分析

基于 MC9S08JM60 实现的手持式超声波测距仪的控制器选用 Freescale 公司基于HCS08内核系列的单片机MC9S08JM60。MC9S08JM60内核频率为48MHz，具有高达60KB片内可编程 FLASH 和 4KB 片内 RAM，内置 USB2.0 控制器、模数转换器 ADC、串行通信接口 SCI、串行外设接口 SPI、$I^2C$ 控制器、定时器等多种外设，方便进行系统设计。

按照图 24-1 所确定的系统原理框图，超声波发射/接收传感器是整个系统的关键，因此首先需选择合适的传感器。

目前，超声波发生器可以分为两大类：一类是用电气方式产生超声波，一类是用机械方式产生超声波。电气方式包括压电型、磁致伸缩型和电动型等；机械方式有加尔统笛、液哨和气流旋笛等。它们所产生的超声波的频率、功率和声波特性各不相同，因而用途也各不相同。目前较为常用的是压电式超声波发生器。

压电式超声波发生器实际上是利用压电晶体的谐振来工作的。超声波发生器有两个压电晶片和一个共振板。当它的两极外加脉冲信号、其频率等于压电晶片的固有振荡频率时，压电晶片将会发生共振，并带动共振板振动，便产生了超声波。反之，如果两电极间未外加电压，当共振板接收到超声波时，将压迫压电晶片作振动，将机械能转换为电信号，这时它就成为超声波接收器了。

本例中选用的超声波传感器为 UCM40 的压电陶瓷传感器，它的工作电压是 40kHz 的脉冲信号，接收头采用与发射头配对的 UCM40R。

## 24.3 硬件电路设计

超声波测距仪系统主要有电源电路、复位电路、仿真调试电路、接收单元电路、发射电路、按键电路、显示电路和单片机电路，下面进行具体介绍。

### 24.3.1 接收单元

接收单元将超声波变送器的输出信号处理后送入单片机的 IO 口，其电路原理图如图24-2 所示。

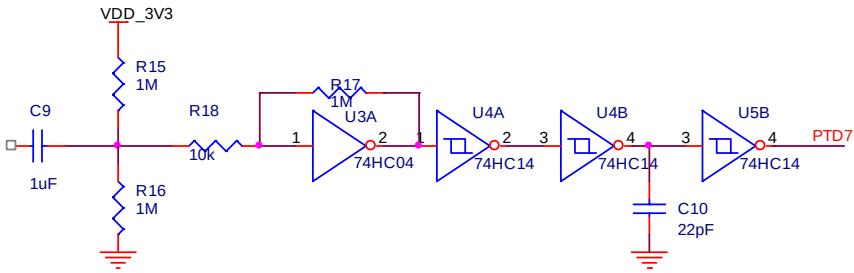



图 24-2　接收单元电路

如图 24-2 所示，超声波变送器的输出经 1uF 电容滤波滤除噪声后，经偏置电路，信号将被置于 3.3V/2 中心附近。74HC04 高速 CMOS 反相器作为线性增益放大器，在此将增益设为 100，将信号放大到适合的 TTL 电平范围内。74HC04 的输出经 74HC14 施密特触发器，将信号转换成标准的方波 TTL 信号，该方波信号经 22pF 滤波电容滤除噪声，输出脉冲送入单片机的 PTD7 引脚。

## 24.3.2　发送单元

发送单元负责产生超声波，由单片机产生一系列的脉冲，脉冲信号放大后发送出去，遇到障碍物则接收单元接收回波。使用高速比较器 LM311 来放大信号，其电路原理如图 24-3 所示。

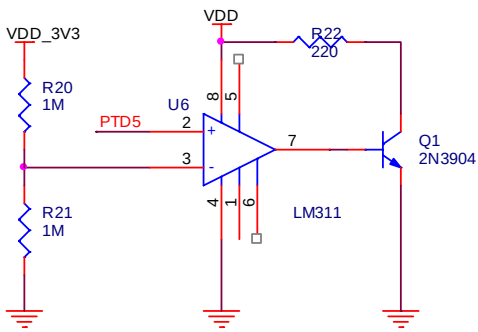



图 24-3　发送电路原理图

## 24.3.3　按键电路

本设计共采用了 3 个独立按键 SW1、SW2、SW3，它们分别接 MC9S08JM60 的 PTD1～PTD3 引脚，按键主要用来处理测距仪工作模式的切换。其电路原理如图 24-4 所示。

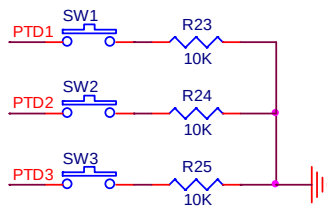



图 24-4　按键电路原理图

### 24.3.4 显示单元

显示单元用来显示测量的结果和工作模式。LCD 选用 LCD1602 液晶模块，8 位总线方式，液晶的 8 位数据线接单片机的 PTB 口，控制线接单片机的 PTC0～PTC2 引脚。如图 24-5 所示为显示单元的电路原理图。

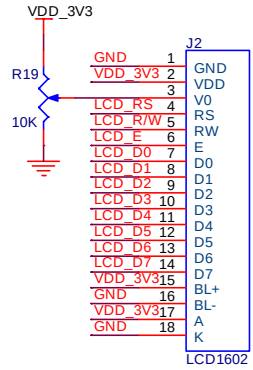



图 24-5 显示单元电路图

### 24.3.5 单片机系统

单片机选用 Freescale 公司的 MC9S08JM60，只需将单片机的 PTB 口、PTC 口、PTD 口与显示电路、按键电路和接收/发送单元电路接口即可。另外单片机还需要相应的振荡电路才能正常工作。如图 24-6 所示为具体的单片机系统电路图。

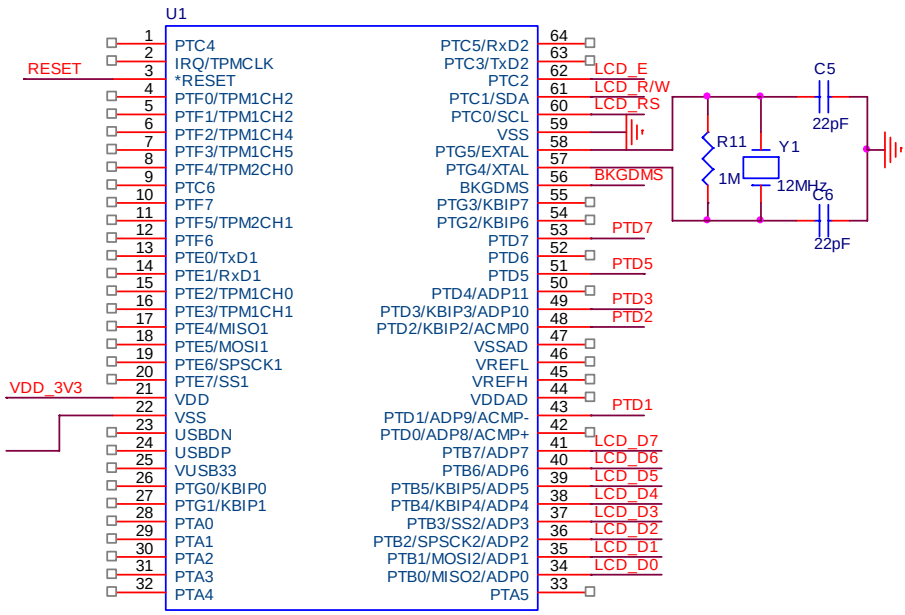



图 24-6 单片机系统电路图

由图 24-6 可以看出，该单片机电路非常简单。考虑到电源的输入纹波对单片机的影响，在电源的管脚上增加一个 104 的滤波电容，以减少干扰。

## 24.4 软件设计与代码

### 24.4.1 软件设计流程

基于 MC9S08JM60 的手持式超声波测距仪软件主要完成两个功能：一是发送 40kHz

的脉冲；二是检测什么时候接收到返回的脉冲。计算发送脉冲和返回脉冲间的时间间隔，从而根据公式计算出距离。系统软件的流程图如图 24-7 所示。

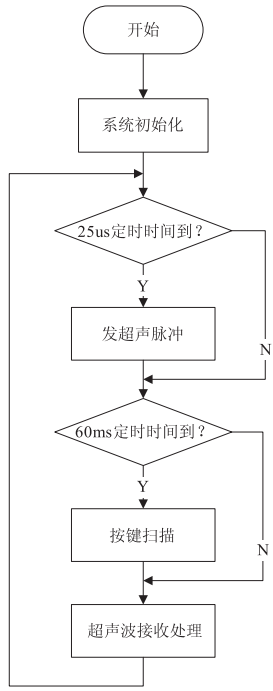



图 24-7　系统软件流程图

如图 24-7 所示，系统初始化完成后，即进入循环，不断查询定时标志，若定时标志有效，则执行相应的子程序。超声波接收处理也在循环内处理。下面对主程序中的子程序进行具体的介绍。

## 24.4.2　超声波发送子程序

超声波发送子程序利用了 MC9S08JM60 单片机的定时器 2。定时器 2 工作在 PWM 方式下，产生 40kHz 的方波信号，产生的方波经 PTD5 管脚发送出去。其具体程序如下：

```
void Ultrasonic_Send_Pulse(void)
{
    if(sendpulse)                  //发送超声波
    {
        received = 0;
        if(count>0)
        {
           setReg8(TPM2SC, 0x48);  //使能定时器
          count--;
        }
```

```
        else
        {
           sendpulse = 0;
          setReg8(TPM2SC, 0x00);    //禁止定时器
          setReg8(TPM1CNT, 0x00);
           ovrflwcount = 0;
        }
    }
}
```

### 24.4.3 超声波接收子程序

超声波接收子程序利用了 MC9S08JM60 单片机的定时器 1，定时器 1 的工作频率为 250kHz。程序利用定时器 1 的定时中断来计算发送超声波和返回超声波之间的时间间隔。下面给出超声波接收的具体程序。

```
void  Ultrasonic_Receive_Process(void)
{
    if((ovrflwcount>500)&&(received==0))
    {
        Lcd_Disp_Str(0,0,"d =");
        received = 1;
        Lcd_Disp_Str(0,4,"out of range");
    }
    //如果管脚为高，则接收
    if( (~Receiver_GetVal())&&(wait == 0)&&(received==0))
    {
        stoptime = TPM1CNT;                    //保存时间节拍数
        stopcount = ovrflwcount;               //接收前已过的时间
        received = 1;

         Lcd_Disp_Str(0,0,"d =");
        ticks = ((stopcount*256)+stoptime);   //计算总节拍数
        distance = (ticks * 136 * (tpm / ctpm)) / 200;//计算距离
        prevdist = currdist;                   //存储以前的距离值
        currdist = distance;                   //保存当前距离值

      Lcd_Disp_Str(0,4,"            ");
      Lcd_Disp_Str(0,4,&lcd_buffer);

        if(normal)
        {
            if(rcount>0)
            {
```

```
            delayedsend = 60000;            //下次脉冲发送前的延时
            sendpulse = 1;
            count = 2*pulsecount;
            wait = waittime;
            distsum = distsum + distance;
            if(rcount == (samples/2))       //取最后一次采样的中值
            {
                speedsum = distance;        //计算速度
            }
            else if(rcount == 1)
            {
                speedsum = speedsum - distance;
            }
            rcount--;
        }
        else
        {
            distsum = distsum/(samples);    //取平均
            speedsum = (speedsum/(samples/100));//计算速度

           Lcd_Disp_Str(0,4,"          ");
           Lcd_Disp_Str(0,4,&lcd_buffer);
           Lcd_Disp_Str(1,0,"spd =  ");
           Lcd_Disp_Str(1,9,&lcd_buffer);
            distsum = 0;                    //变量清零
            speedsum = 0;
            normal = 0;
        }
    }
    else if(calibrate)
    {
        if(rcount>0)
        {
            delayedsend = 60000;
            sendpulse = 1;
            count = 2*pulsecount;
            wait = waittime;
            distsum = distsum + distance;   //累加距离值
            tsum = tsum + ticks;            //累加节拍值
            rcount--;
        }
        else
        {
            distsum = distsum / samples;    //距离平均
```

```
                tsum = tsum / samples;          //节拍数平均

                Lcd_Disp_Str(0,4,"          ");
               Lcd_Disp_Str(0,4,&lcd_buffer);
               Lcd_Disp_Str(1,0,"avgtick =  ");
               Lcd_Disp_Str(1,10,&lcd_buffer);
                distsum = 0;
               ctpm = tsum;
               tsum = 0;
              calibrate = 0;
            }
        }
    }
}
```

### 24.4.4 按键扫描子程序

按键扫描子程序主要处理测距仪三种工作模式的切换。三种工作模式为：正常模式、改变采样数模式和校正模式。在正常工作模式下，按下按键则发送超声波，单片机则记录下发射波和返回波之间的时间节拍数，根据此节拍数，计算出发射点和障碍物间的距离，计算结果显示在 LCD 上，同时还计算出速度。在改变采样数模式下，可以改变采样的次数，但采样次数需大于 50 次，以确保较精确的测量结果。由于海拔和湿度会影响声音的传播速度，导致声速在不同海平面上有区别，为确保准确测量，需通过校正来消除此影响。下面给出按键扫描子程序的具体程序。

```
void Key_Process(void)
{
    switch(pushstate)
    {
        case nopush:
            // 有按键按下?
            if((~Key1_GetVal()) || (~Key2_GetVal()) || (~Key3_GetVal()))
            {
                pushstate = maybepush;
            }
            else
            {
               pushstate = nopush;
            }
            break;
        case maybepush:
            if((~Key1_GetVal()) || (~Key2_GetVal()) || (~Key3_GetVal()))
            {
```

```
            pushstate = pushed;
        }
        else
        {
           pushstate = nopush;
        }
       break;
    case pushed:
        if((~Key1_GetVal()) || (~Key2_GetVal()) || (~Key3_GetVal()))
        {
            pushstate = pushed;
            if(~Key1_GetVal())
            {
               key1 = 1;            //按键 1 按下
            }
            else if(~Key2_GetVal())
            {
               key2 = 1;            //按键 2 按下
           }
           else if(~Key3_GetVal())
           {
               key3 = 1;           //按键 3 按下
          }
        }
        else
        {
           pushstate = maybenopush;
        }
       break;
    case maybenopush:
        if((~Key1_GetVal()) || (~Key2_GetVal()) || (~Key3_GetVal()))
        {
            pushstate =  pushed;
        }
        else
        {
            if(received)
            {
              if(!mode)
              {
                 if(key1)
                 {
                     sendpulse = 1;   //发送一个脉冲
                     count = 2*pulsecount;
```

```
            wait = waittime; //等待接收脉冲
            rcount = samples;//采样次数
            key1 = 0;
            normal = 1;      //正常操作模式标志位
        }
        else if(key2)
        {
            ctpm = 73;       //复位校正
            mode = 1;        //切换到采样模式标志位

            Lcd_Disp_Str(1,0,"b1- ");
            Lcd_Disp_Str(1,0,"b2+ b3 back");
            key2 = 0;
        }
        else if(key3)
        {
            sendpulse = 1;  //发送脉冲
            count = 2*pulsecount;
            wait = waittime;
            calibrate = 1;  //校正模式标志位
            key3 = 0;
            rcount = samples;
        }
    }
    else if(mode)          //为改变采样数模式下
    {
        if(key1)           //按键 1 按下?
        {
            if(samples>50) //如果采样次数大于 50，则减少采样次数
            {
                samples = samples - 50;
            }
            key1 = 0;
          Lcd_Disp_Str(0,0,"      ");
            Lcd_Disp_Str(0,0,&lcd_buffer);

        }
        else if(key2)    //按键 2 按下?
        {
            samples = samples+50;//增加采样次数
         Lcd_Disp_Str(0,0,"       ");
            Lcd_Disp_Str(0,0,&lcd_buffer);
            key2 = 0;
        }
```

```
                    else if(key3)
                    {
                        ctpm = 73; //复位校正
                        mode = 0;  //设置模式为改变采样数模式
                       Lcd_Disp_Str(0,0,"b1 sendpulse");
                       Lcd_Disp_Str(1,0,"b2 mode2 ");
                       Lcd_Disp_Str(1,0,"b3 cal");
                        key3 = 0;
                    }
                }
            }
        }
        break;
    }
}
```

## 24.4.5　测距仪主程序

测距仪主程序的设计基于事件驱动的编程方式。主程序首先查询两次发送延时时间是否到，若到了，则启动超声波发送子程序；按键扫描时间为每 30ms 执行一次，若 30ms 定时时间到，则执行按键扫描子程序；最后调用超声波接收处理子程序，对接收的返回波进行处理，利用数学公式，计算发射点和障碍物间的距离。下面给出手持式超声波测距仪主程序的具体程序。

```
void main(void)
{
  PE_low_level_init();
  System_Initialize();

  for(;;)
  {
    if(delayedsend == 0)     //两次发送延时时间到?
    {
        if(time1 == 0)
        {
            Ultrasonic_Send_Pulse();
            time1 = t1;
        }
    }
    if(time2 == 0)          //30ms 定时时间到?
    {
        Key_Process();
        time2 = t2;
    }
```

```
        Ultrasonic_Receive_Process();//超声波接收处理
      }
    }
```

## 24.5 实例总结

本章详细介绍了基于 MC9S08JM60 单片机的手持式超声波测距仪的硬件设计和软件设计，并给出了软件系统的各个模块，最后给出了主程序。实例充分利用 MC9S08JM60 丰富的外设资源，从而简化了硬件设计。在介绍软件设计时，详细介绍了各个模块的功能。经实际测试证明，该测距仪工作稳定，能满足一般近距离测距的要求，且成本较低、有良好的性价比。本系统实现的功能具有通用性，读者可以在本章的基础上，设计更加复杂的系统，如将本章所实现的功能和智能机器人相结合，可以很容易地实现机器人的避障功能；将本章实现的功能简化，也很容易实现汽车的倒车雷达功能。